2015-11

An Exploration Of The Effects Of Enhanced Compiler Error Messages For Computer Programming Novices

Brett A. Becker
Technological University Dublin

An Exploration of the Effects of Enhanced Compiler Error Messages for Computer Programming Novices

A thesis submitted to Dublin Institute of Technology in part fulfilment of the requirements for the award of Masters (M.A.) in Higher Education

by

Brett A. Becker

November 2015

Supervisor: Dr Claire McDonnell

Learning Teaching and Technology Centre, Dublin Institute of Technology

Declaration

I certify that this thesis which I now submit for examination for the award of Masters (M.A.) in Higher Education is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my own work.

This thesis has not been submitted in whole or in part for another award in any other third level institution other than part-fulfilment of the award named above.

The work reported on in this thesis conforms to the principles and requirements of Dublin Institute of Technology's guidelines for ethics in research.

Dublin Institute of Technology has permission to keep, lend or copy this thesis in whole or in part, on condition that any such use of the material of the thesis be duly acknowledged.

Signed..... Date.....

Abstract

Computer programming is an essential skill that all computing students must master and is increasingly important in many diverse disciplines. It is also difficult to learn. One of the many challenges novice programmers face from the start are notoriously cryptic compiler error messages. These report details on errors made by students and are essential as the primary source of information used to rectify those errors. However these difficult to understand messages are often a barrier to progress and a source of discouragement. A high number of student errors, and in particular a high frequency of repeated errors – when a student makes the same error consecutively – have been shown to be indicators of students who are struggling with learning to program.

This instrumental case study research investigates the student experience with, and the effects of, software that has been specifically written to help students overcome their challenges with compiler error messages. This software provides help by enhancing error messages, presenting them in a straightforward, informative manner. Two cohorts of first year computing students at an Irish higher education institution participated over two academic years; a control group in 2014-15 that did not experience enhanced error messages, and an intervention group in 2013-14 that did.

This thesis lays out a comprehensive view of the student experience starting with a quantitative analysis of the student errors themselves. It then views the students as groups, revealing interesting differences in error profiles. Following this, some individual student profiles and behaviours are investigated. Finally, the student experience is discovered through their own words and opinions by means of a survey that incorporated closed and open-ended questions.

In addition to reductions in errors overall, errors per student, and the key metric of repeated error frequency, the intervention group is shown to behave more cohesively with fewer indications of struggling students. A positive learning experience using the software is reported by the students and the lecturer. These results are of interest to educators who have witnessed students struggle with learning to program, and who are looking to help remove the barrier presented by compiler error messages.

This work is important for two reasons. First, the effects of error message enhancement have been debated in the literature – this work provides evidence that there can be positive effects. Second, these results should be generalisable at least in part, to other languages, students and institutions.

Acknowledgements

I would like to acknowledge the professional guidance, support and amazing patience of my supervisor Dr. Claire McDonnell, and Dr. Jen Harvey for serving as second reader. I would also like to thank all of the staff of the Dublin Institute of Technology Learning, Teaching and Technology Centre for their constructive suggestions and enthusiastic encouragement during the past two very enjoyable years.

I am very appreciative for the support of the president of the College of Computing Technology, Neil Gallagher, and the extremely helpful assistance of Ricardo Iwashima and (soon to be Dr.) Graham Glanville. I am also indebted to Dr. Fiona O'Riordan for proofreading and many enjoyable conversations, not only limited to this work.

Finally I am very thankful to my mother Sharon, who has read every word in now four theses, and who had to endure way too much talk about them. This is a similar plight faced by my fiancée Dr. Catherine Mooney, who I would like to thank for everything, but will only cite here her amazing patience and skill in data analysis. Without her help this work would only be a shadow of what it has become.

Contents

List of Figures

List of Tables

Abbreviations and Definitions

ACM – The Association of Computing Machinery, the world's largest scientific and educational computing society (www.acm.org).

API – Application Program Interface: A set of routines, protocols, and tools for building software applications.

CEM – Compiler Error Message: A message generated by a compiler which is intended to bring to the attention of the programmer the location, nature and cause of a violation of the syntax and sometimes semantics of the programming language in code the programmer has written.

CP – Computer Programming

CS – Computer Science

CS1 – Computer Science 1: In computer science education research literature, the introductory programming module is often referred to as CS1 regardless of institution, language or local title.

CSE – Computer Science Education

CSER – Computer Science Education Research

ECEM – Enhanced Compiler Error Message: A compiler error message which has been enhanced by some means, in an effort to bring more clarity to the location, nature and/or cause of the syntactic/semantic violation than the original compiler error message provides.

ICT – Information Communications Technology

IDE – Integrated Development Environment: An editor used to write, compile and execute computer programs.

Java – A popular, modern, object oriented, programming language.

javac – The Java compiler, which comes with the Java development kit.

JDK – Java Development Kit – A collection of programming tools and a runtime environment, necessary to compile and execute Java programs.

PC – Principal Component: A set of linearly uncorrelated variables resulting from a principal component analysis.

PCA – Principal Component Analysis: A non-parametric method of reducing a complex data set to reveal hidden, simplified dynamics within it by converting a set of observations of variables (which may be correlated) into a set of values of linearly uncorrelated principal components.

SIGCSE – the ACM Special Interest Group on Computer Science Education (www.sigcse.org). The annual SIGCSE Technical Symposium is commonly called simply SIGCSE also (www.sigcse.org/events/symposia).

Chapter 1 Introduction

"If a compiler error was worth one Euro, I would be a millionaire."

– Anonymous CS1 student, from this study, in questionnaire about compiler error messages

This thesis investigates one of the many difficulties novice programming students encounter when learning computer programming (CP) – compiler generated error messages (CEMs). Some existing systems which help novice programmers provide enhanced compiler error messages (ECEMs) as a feature, however ECEMs have not often been studied rigorously or in isolation.

Computer programming is a process which takes an original problem formulation and turns it into a program executed on a computer, which automates finding a solution to the problem. Programming is an expected outcome of a computer science (CS) student's education (McCracken, et al., 2001) and a core competency for many in the IT industries (Orsini, 2013).

Programs written in most modern computer languages are typed into an editor, and the resulting program is then *compiled*. Compiling a program is complex, and handled by software colloquially called *a compiler*. This study focusses on Java, currently the most popular programming language for teaching novices to program (Davies, Polack-Wahl, & Anewalt, 2011; Siegfried, Greco, Miceli, & Siegfried, 2012), and consistently one of the most popular languages used in industry (Programming Language Popularity, 2013; Cass, 2015; TIOBE Software, 2015). It should be noted that Java and its popularity as an introductory programming language choice is not without critics (Sigfried & Chays, 2008), and that Python has increasingly grown in popularity as an introductory language recently (Guzdial, 2011; Guo, 2014). As many of the students involved in this study are non-native English speakers, it should also be noted that in general non-native English speakers 'program in English' (that is with English keywords, etc.), as almost all programming languages are 'written in' and use English exclusively. For a brief discussion on this, please see Appendix A.

CEMs and the difficulties they present to students have been pervasive themes in teaching beginners the subject of CP for several decades. In 1976, 'cryptic diagnostics'

was cited as one of the primary things that make programming difficult, for students in particular (Wexelblat, 1976). Thirty years after Wexelblat's comments, Jadud noted that "the error messages generated by commercial compilers are often uninformative and sometimes misleading" (2006, p. 16). Unfortunately these characteristics are still indicative of the CEMs provided by modern languages now nearly forty years later – cryptic and uninformative, often terse and misleading. Just last year Ko wrote a blog post titled "Programming languages are the least usable, but most powerful human-computer interfaces ever invented", calling a particular PHP CEM "inscrutable" (Ko, p. 1). Mark Guzdial wrote a blog post on the ACM website two days later with the same title but adding "and learnable" after "usable" (2014). It is amazing that the comments in this paragraph span nearly 40 years and are not just talking about something that is difficult; they are talking about a reason *why* that something is difficult. We have known for a long time that CEMs can make programming difficult. The question is: Why are *compiler error messages* still a problem?

In 2004, an ITiCSE[1] working group established that problems introductory programming students encounter are not limited to single institutions, teaching methodologies or styles, or even particular languages (Lister, et al., 2004). Indeed Wexelblat's 1976 study was not specific to a particular language. This is important, as most studies do focus on one particular language; however the lessons learned have the potential to be transferred to students of other languages at other institutions.

Most high-level programming languages have their own set of vocabulary (keywords) and grammar (syntax). Compilers identify keywords and enforce syntax. For Java, the compiler is called *javac* (pronounced *java-see*, or less frequently *javack*), part of the Java Development Kit (JDK). If a Java program in any way violates the syntax[2] of the language, which is defined in the Java Language Specification (Oracle Corporation, 2015), details on violations are returned as CEMs designed to help the programmer locate, identify and rectify them. There is experimental evidence that Java has a syntax which is no better than a language with randomly chosen keywords (a devised 'placebo' language called Randomo), and that Java is less intuitive and easy to use for novices than languages such as Ruby, Python and Quorum (Stefik & Siebert, 2013). This is also

[1] Annual conference on Innovation and Technology in Computer Science Education, a conference of SIGCSE, the ACM Special Interest Group on Computer Science Education (www.sigcse.org/events/iticse).
[2] Some semantic errors also cause compiler error messages in Java.

evidence that Java CEMs are particularly difficult for novices, and means of learning how to deal them are needed. Ben-Ari (2007, p. 7) said of Java CEMs: "… there is no easy way to find the exact cause of such errors except by checking the code around the location of the error character by character looking for the syntax error."

1.1 Context

1.1.1 Impact of Compiler Error Messages on Novice Programmers

CEMs, their effectiveness, and difficulties with them have been a problem for novice programmers and those who teach them for decades. CEMs are one of the most important tools that a language offers its programmers, and for novices, their feedback is especially critical (Marceau, Fisler, & Krishnamurthi, 2011b). Kummerfield and Kay note: "Syntax error correction is the first step in the debugging process. It is not possible to continue program development until the code compiles. This means it is a crucial part of the error correction process." (2003, p. 109). However, novice programmers have been shown in several studies to have trouble interpreting CEMs (Hristova, Misra, Rutter, & Mercuri, 2003; Hartmann, MacDougall, Brandt, & Klemmer, 2010; Traver, 2010), which prevents them from fixing the errors in their programs – something that is obviously detrimental to the process of learning to program successfully (Jadud, 2006; Nienaltowski, Pedroni, & Meyer, 2008).

When a syntax error occurs in a computer program it is similar to a grammatical or spelling error being detected in a word-processing document, and the following are necessary for the error to be corrected:

1. Error is identified by the compiler
2. Compiler reports that error has occurred, presenting CEMs, typically containing:
    a. Error location
    b. Error type or nature
    c. Additional details to aid the programmer in rectification
3. Programmer uses information from 2 to rectify the error

When a breakdown occurs, preventing one of the above steps from completing successfully, the error will not be rectified. Compilers are based on robust, deterministic algorithms. They do not miss errors, and do not forget to report them, or information about them. When dealing with CEMs, the breakdown *always* occurs in step 3, when

the human programmer cannot fix the error based on the information provided in the CEM. This leads us to one of three possibilities – there is something wrong with the human interpreting the information, or something wrong with the CEMs or perhaps something wrong with both.

Figure 1.1 demonstrates an example of this breakdown with a small Java program containing two syntax errors (a), followed by the resulting CEMs designed to expose/explain these errors (b), and an English-language explanation of what the actual errors are (c).

(a) program with two syntax errors	`public class hello {`
	`  public static void main(string[] args) {`
	`  system.out.println("Hello World!");`
	`  }`
	`}`

(b) resulting CEMs which result from the errors in (a)	`C:\Users\Brett\Desktop\junk\hello.java:2: error: cannot find symbol`
	`  public static void main(string[] args) {`
	`  ^`
	`  symbol:  class string`
	`  location: class hello`
	
	`C:\Users\Brett\Desktop\junk\hello.java:3: error: package system does not exist`
	`  system.out.println("Hello World!");`
	`  ^`
	`2 errors`
	`Process Terminated ... there were problems.`

(c) English-language explanation of the errors in (a)	`The “s” in “string” on line 2 should be capitalised.` `The “s” in “system” on line 3 should be capitalised.`

Figure 1.1 Java program with two syntax errors (a), resulting CEMs (b), and English-language description of the errors (c).

It is easily understandable that the CEMs would not necessarily lead a novice to easily determine the cause of the actual errors.

CEMs present an interesting and somewhat rare pedagogical situation from the students’ and teachers’ perspectives in that the feedback supplied by compilers is immediate, consistent, detailed and informative (Robins, Rountree, & Rountree, 2003). This is despite them being of questionable use by the novice as even ‘simple’ errors are often technically phrased, and can be confusing to beginners as they are often aimed at expert users (Reis & Catrwright, 2004; Nienaltowski, Pedroni, & Meyer, 2008).

Additionally, the level of exactness demanded by a compiler is arguably unprecedented outside the world of computing (particularly in that even from an absolute beginner such exactness is required). Gries illustrated this succinctly early on (1974, p. 83):

> *Programming requires exactness and precision unknown in many other fields. Even in a mathematics paper, syntax errors and many logical errors can be understood as such and mentally corrected by the reader. But a program must be exact in every detail.*

This requirement for exact precision has been shown to be a common source of frustration for novices (Rogerson & Scott, 2010).

1.1.2 My Own Context

In my own practice I have made several observations which sparked my interest in this research:

1. Some students are confounded by compiler error messages and do not directly correlate them with errors in their code.
2. Some students ask for help on a particular CEM multiple times. It seems that they are not *learning* from CEMs – instead they see them as hindrances, blocking them from completing the task at hand.
3. The way that CEMs are presented vary in usefulness, clarity and arguably correctness – although they are algorithmically correct, to a novice they sometimes *seem* wrong.

In a departure from many of the studies on novice interaction with CEMs, this research seeks to determine if ECEMs can benefit the learner in a controlled, empirical manner.

1.2 Rationale and Motivation

In addition to the specific observations I have made in my own practice, I also have concerns about computer science education in general. As a computer science educator with over a decade of practice, I have seen the effects of high dropout rates, poor performance (particularly in programming modules), and have witnessed programming incite fear, apprehension and sometimes disdain in students. Disturbingly the causes of these forces remain unclear globally, yet must be under our noses. After all, the students, the curriculum, the pedagogy, etc. are all beneath our own institutional roofs. Why is it so difficult to pin down and address the causes of these worries? This work seeks not to

answer this question sweepingly and convincingly – the myriad efforts over decades have yet to answer this question. This work does seek to understand the role of CEM enhancement in the hopes of improving the process of learning to program, and hopefully having a (small) knock-on effect on retention, and the overall student experience.

In addition to my immediate observable concerns, I am concerned about higher education information and communication technologies (ICT) courses having poor retention rates domestically and globally (Caspersen & Bennedsen, 2007; Peters & Pears, 2012). Recently the subject of CS had the largest dropout rate of all courses in the UK, and approached 60% in Finland (Matthíasdóttir & Geirsson, 2011). In Ireland, ICT courses have the highest risk for dropout among all third-level programmes, and suffer a 27% non-presence rate (Mooney, Patterson, O'Connor, & Chantler, 2010). In the United States, the average annual attrition rate for CS majors has been reported to be 19%, and up to 66% at some schools (Sloan & Troy, 2008).

In CS education research (CSER) literature, the introductory programming module is referred to as CS1 regardless of institution, language or local title (Hertz, 2010). As early as 1970, teaching CS1 was proving problematic. Fenichel, Weizenbaum and Yochelson noted, "A problem confronting the nation's colleges and universities is that of providing instruction in elementary computer programming" (1970, p. 141). CS1 itself often has high dropout/failure rates (Robins, Rountree, & Rountree, 2003; Beaubouef & Mason, 2005; Piteira & Costa, 2011; Yadin, 2011; Porter, Guzdial, McDowell, & Simon, 2013; Watson & Li, 2014) and is cited as one of the primary factors in the overall dropout rate of ICT programmes (McCracken, et al., 2001; Fenwick, Jr., et al., 2009; Shuhidan, Hamilton, & D'Souza, 2009; Vogts, Calitz, & Greyling, 2010; Matthíasdóttir & Geirsson, 2011). Evidence has even shown that students 'hate programming' and vow their future careers will not involve it (Thomas, Ratcliffe, Woodbury, & Jarman, 2002).

Within the subject of CS1, the unavoidable topic of CEMs has been shown to be a barrier to successful outcomes, and although there has been research in this arena for decades, and has picked up pace in the past ten to fifteen years (Allen, Cartwright, & Stoler, 2002; Hristova, Misra, Rutter, & Mercuri, 2003; Flowers, Carver, & Jackson, 2004; Hartmann, MacDougall, Brandt, & Klemmer, 2010; Carter & Blank, 2013; Stefik & Siebert, 2013). However research in this is notably limited and it remains a niche

area. Nonetheless, it is important, as links can be drawn between CEMs and performance in programming (Tabano, Rodrigo, & Jadud, 2011) and from there to high dropout rates, often 'through' the subject of CS1. Difficulties with CEMs are only one piece to this puzzle however, with many issues at play which combined result in poor programming performance and ultimately poor retention seen in CS programmes (Bergin & Reilly, 2005), as shown in Figure 1.2.

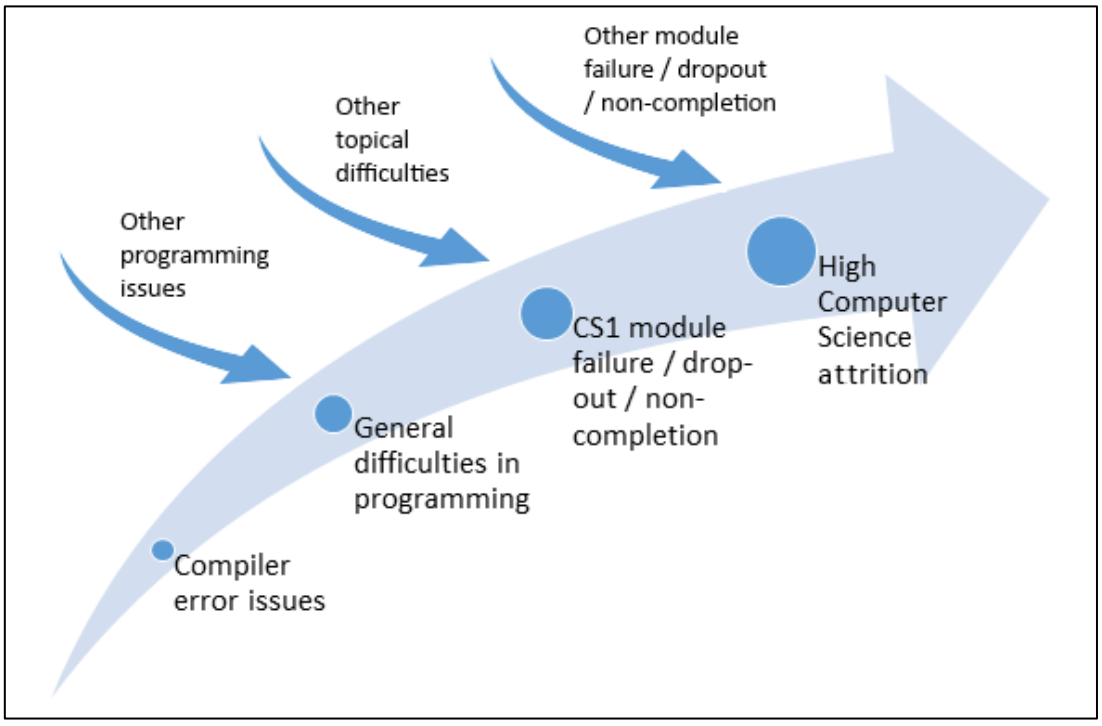

Figure 1.2 Link between compiler error message issues, general difficulties in programming, CS1 module failure / dropout and high computer science attrition rates.

In addition, it has been shown that even for intermediate students, industrial Java Integrated Development Environments (IDEs)[3] are unsuitable (van Tonder, Naudé, & Cilliers, 2008), yet their use is common. Compared to using professional IDEs, pedagogical environments are beneficial to the perceptions of novice programmers learning to program, specifically their feelings of achievement and learning (Vogts, Calitz, & Greyling, 2010). This justifies the motivation to develop a specific pedagogical tool such as the one used for this study, to help novices interpret and effectively utilise CEMs. In addition, significant motivation for this study stems from

[3] An IDE is essentially an editor which a programmer uses to write programs, much like one uses Microsoft Word to write documents.

a similar study published halfway through this work (Denny, Luxton-Reilly, & Carpenter, 2014) and subsequent discussions it created in the CSER community.

1.3 Research Aims & Objectives

The aim of this research is to answer the research question:

Do enhanced compiler error messages help students who are learning to program?

In investigating this question, the following sub-questions will be answered:

1. *Do enhanced compiler error messages reduce the overall number of errors?*
2. *Do enhanced compiler error messages reduce the number of errors per student? If so, for what compiler error messages in particular?*
3. *What effect do enhanced compiler error messages have on students, particularly students struggling with programming?*
4. *Do enhanced compiler error messages reduce the number of repeated errors?*
5. *What are learner and educator views on enhancing compiler error messages?*

This research utilises a Java editor called Decaf, specifically written for this research by the author. Decaf uses available information (the erroneous line of code and the CEM which has been intercepted) and attempts to construct more specific and helpful ECEMs which are presented to the user, along with the original CEMs. The aim is to help students rectify their errors more effectively, while providing a side-by-side opportunity to get used to and learn the actual meanings of the original, often cryptic CEMs.

Specific objectives are measuring and comparing the following metrics between a control group who used Decaf in 'pass-through' mode (with no ECEMs) and an intervention group who used Decaf in 'normal' mode (with ECEMs):

- Total number of CEMs
- Frequency of CEMs per student, including specific CEMs
- Number of repeated (consecutive) CEMs per student

This research also investigates group and individual error profiles, and triangulates the above quantitative data with student questionnaires and lecturer interviews.

Any evidence that ECEMs help students in programming could contribute to improved learning and academic performance, better CS1 results, and ultimately lower CS attrition rates. Stefik & Siebert stated this so (2013, p. 2):

> *…since novices must become comfortable with syntax in order to use a general purpose programming language, trying to identify where syntactic barriers exist may benefit instructors teaching various technologies. If the barriers can be identified in a specific enough way (e.g., which tokens, words, or symbols should be altered?) this may also benefit students in the long run as programming languages evolve.*

1.4 Thesis Layout

This thesis is laid out as follows: Chapter 2 provides a literature review on the challenges of learning to program and work to-date in novice experience with CEMs, particularly in the Java language. Chapter 3 sets out the research methodology, including theoretical perspective, specific methods, limitations and delimitations, design decisions, and technical details of the Decaf editor. Chapter 4 presents findings, analysis and discussion. Chapter 5 sets out conclusions, implications for practice, and directions for future work.

Chapter 2 Related Work

"... the error messages take brevity to the extreme. There are only four messages, and each consists of two words, one of which is ERROR. Indeed, the response of the APL system to a program error is reminiscent of the mourner on an Irish funeral procession, who on being asked 'who's dead?' replied 'The man in the box up front'."

- D. W. Barron, 1976 as cited in (Wexelblat, 1976, p. 336)

Work on compiler error messages has a history of over 40 years. Outside of some general trends which have remained (such as compiler error messages causing difficulties), the constantly evolving landscape of programming and programming languages demands ever more current studies. This results in a never truly complete picture of the area. This chapter presents related work in compiler error messages under the following five subsections.

- Compiler Error Messages: Importance and Difficulties
- Patterns of Behaviour and Links to Performance
- Frequency, Categorisation and Databases of Errors
- Compiler Error Enhancement
    - Systems for Teaching Novices
    - Industrial Systems
- The Current Challenge

2.1 Compiler Error Messages: Importance and Difficulties

CEMs, their effectiveness, and the difficulties they present to students have been pervasive themes in CSER for several decades, regardless of programming language. Early on it became evident that in some cases CEMs were not adequate for serving their critical purposes. In 1970, Fenichel, Weizenbaum and Yochelson identified that "once an error is detected, it is to be pointed out to the student in as unambiguous a fashion as possible" (p. 142). Koster stated that one of the core tasks of a compiler is to check whether the source program is correct and if it is not, to give all useful information for correcting it (1973), and particularly in minimal time (Graham & Rhodes, 1973). Wexelblat (1976) observed that most compiler error messages reflect what the program

did, not what the user did – in other words the message states the effect of the error, not the cause. Exceptionally terse messages were also cited as troublesome. Litecky and Davis (1976) investigated student CEMs in the COBOL language, determining their feedback was not optimal for users, particularly for those learning (1976, p. 33). The appearance of such straightforward statements on CEMs early in the literature indicate their importance and the difficulties they pose, particularly for beginners.

As CSE became more widespread, Pascal secured its position as the first dominant programming language for teaching. Brown (1983) investigated issues with CEMs in Pascal, finding them to be inadequate, and Chamillard and Hobart (1997) addressed concerns over syntax errors in their transition from Pascal to Ada97. The title of Kummerfield and Kay's paper investigating CEMs in C, *The neglected battlefields of syntax errors* (2003) gave insight into the growing importance of the area. Bergin, Agarwal and Agarwal (2003) pointed out numerous issues with the C++ language in its use as a teaching language, many of them to do with CEM issues. C++ was a dominant teaching language of its time (taking the lead from Pascal) and eventually replaced by Java, which saw an increase in research into CEMs.

Good CEMs are critical for novice programmers, and play at least two important roles: as a programming tool they should help the user progress towards a working program, and as a pedagogic tool they should help the user understand the problem that led to the error (Marceau, Fisler, & Krishnamurthi, 2011a; Marceau, Fisler, & Krishnamurthi, 2011b). They should avoid frustrating the user, and not lead the user down the wrong path to correcting the problem. I have told students that CEMs are their 'friends' – they are the only front-line information they have to fix their code. However, dealing with CEMs is often a frustrating experience for students (Flowers, Carver, & Jackson, 2004; Hsia, Simpson, Smith, & Cartwright, 2005). Jadud goes as far as stating that compilers are "veritable gold mines for cryptic and confusing error messages" (2006, p. 1), while Traver describes Java errors in particular as "undecipherable" (2010, p. 4). When Motil and Epstein created JJ, a subset of Java, they noted that "Java was still "lurking" in the background, providing weird error messages" (n.d., p. 2). Ben-Ari noted that educators resorted to writing supplementary material to help explain CEMs (2007), while McCall and Kölling stated: "Compiler error messages … are still very obviously less helpful than they could be" (2014, p. 2589). Nielsen put it simply: "Error messages should be expressed in plain language (no codes), precisely indicate the problem, and

constructively suggest a solution" (1994, p. 30). Marceau, Fisler and Krishnamurthi summed up the situation nicely (2011b, p. 3):

> *Yet, ask any experienced programmer about the quality of error messages in their programming environments, and you will often get an embarrassed laugh. In every environment, a mature programmer can usually point to at least a handful of favourite bad error responses. When they find out that the same environment is being used by novices, their laugh often hardens.*

Disturbingly, these statements are very similar to those made in the 1970s.

CEMs also pose problems for educators, particularly in the context of instructor-led or supported laboratory sessions. Coull (2008) identified that tutors spend large amounts of time solving trivial syntactic problems and that time spent with any individual student may be substantial, and the time other students must wait for help is therefore extended. In addition students tend to make mistakes similar to those of their peers at similar stages, and tutors find themselves solving the same problems for several individuals independently. Denny, Luxton-Reilly & Carpenter noted: "As educators, we have a limited amount of time to spend with each student so providing students with automated and useful feedback about why they are getting syntax errors is very important." (2014, p. 278).

It is appropriate to end this subsection with what was the first effort to design good error messages from the ground up. Michael Kölling's PhD thesis (1999) and prior work (Kölling, Koch, & Rosenburg, 1995; Kölling & Rosenburg, 1996a; Kölling & Rosenburg, 1996b; Rosenburg & Kölling, 1997) introduced a programming language (and IDE) called Blue, designed to avoid the aspects of object-oriented languages and environments which affect their suitability for first year teaching. Blue was carefully constructed so that CEMs would be comprehensible to beginners, avoiding jargon and giving precise information as to the source of the problem, as Kölling explained (1999, p. 146):

> *The quality of messages that a compiler can produce is significantly influenced by the grammar of the language itself and the compiler technology used. In languages like C++, so many ambiguities exist that it is impossible to avoid producing very general or misleading*

> *messages. Blue has been carefully designed to provide a type of grammar and a degree of redundancy within the grammar that enables the generation of good error messages in most cases.*

Students enter into a number of patterns of behaviour when learning to program, and when they are confronted with CEMs. There has been a significant amount of research into these patterns of behaviour, and how these affect student academic performance. This is discussed in the following subsection.

2.3 Patterns of Behaviour and Links to Performance

Dealing with CEMs has been shown to take up a significant amount of student time, and can be a barrier to learning, keeping students mired in syntax, rather than learning semantics (Jadud, 2006) and even 'fighting the compiler', and thinking that programming is "just about getting the syntax right" (Vogts, Calitz, & Greyling, 2010, p. 53). This has an effect on the students' academic performance.

Perkins, Hancock, Hobbs, Martin, & Simmons (1986) noted that, under normal instructional circumstances, some students learn programming much better than others. Investigations of novice programmer behaviour suggest that this happens in part because different students bring different patterns of learning to the programming context. Students often fall into varying combinations of disengaging from the task whenever trouble occurs, neglecting to track closely what their programs do by reading back the code as they write it, trying to repair buggy[4] programs by haphazardly tinkering with the code, or having difficulty breaking problems down into parts suitable for separate chunks of code (Perkins et al., 1986). These authors categorised programming students into groups: *stoppers*, who quickly gave up when faced with errors; *movers*, who would work their way through, around or away from errors; and *tinkerers*, who poke, tweak, and otherwise manipulate their code in a variety of small ways, sometimes making progress towards a working program, sometimes not.

Ahmadzadeh, Elliman and Higgins (2005) also researched the patterns that novices fall into while dealing with CEMs. They looked at the combination of the errors beginners generated and observations of their debugging activities, to try and gain insight into their students' patterns. Their students were classified as either good or weak debuggers

[4] A *bug* is an error, defect or flaw in a computer program that causes undesired and unexpected behaviour.

and good or weak programmers, and noted that being a good debugger did not necessarily make a good programmer and vice versa, although there were strong correlations.

Jadud studied patterns of behaviour in detail in his doctoral dissertation (2006). Although in trying to fix CEMs, students can enter into a number of behaviours, all of these behaviours stem from the same deterministic process. It is these behaviours, which Jadud systemises and quantifies with his *edit-compile-run* cycle and *error quotient*, to better understand the patterns that students experience when dealing with CEMs. The edit-compile-run cycle is a pattern of behaviour that students fall into when dealing with CEMs, and can take many forms, each with identifiable characteristics. A student's error quotient (EQ) is a value, averaged over a session, which reflects whether students are dealing quickly and efficiently with syntax errors (a low EQ), or if they are stuck, making changes that do not actually fix the syntax error in question, and possibly introducing new ones (a high EQ). This quantity provides a powerful indicator for how much or little a student is struggling with the language while programming. Jadud puts the current state of the interactions between programmers and compilers thus (p. 7):

> *…poorly understood. There have been several, paradigmatic shifts in programming languages since the 1950's – from assembly, to procedural/structured programming, to object-oriented programming and design—yet the edit-compile-run cycle remains. Despite this constancy, there has been little systematic research regarding the edit-compile-run cycle.*

Jadud (2006) investigated the link between EQ and student performance on programming assignments, final (written) exams, and overall module marks. Although some correlations were found to exist they were weak, and the overall conclusion was that EQ and academic performance are related, but exactly how remains to be seen. However, Rodrigo et al. (2009), with Jadud as a co-author, found that test scores could be predicted with simple measures such as the student's average number of errors, number of pairs of compilations in error, number pairs of compilations with the same error, pairs of compilations with the same edit location and pairs of compilations with the same error location. This study clearly linked compiler behaviour to performance, but the mechanisms at work, and whether this was just a special case, warrant further research.

The links between compiler error message difficulties, general programming difficulties, and ultimately poor CS retention rates may be evident amongst the CSER community, but more study is needed in this area. This research contributes to this at the root level of compiler error message difficulties.

2.2 Frequency, Categorisation and Databases of Errors

Work on CEMs is impacted significantly by the particular errors that novice students make and the CEMs they are confronted with. A large amount of work has gone into discovering what these errors are and what frequencies they occur at, as well as their categorisation. Recent times have seen these errors being stored and made available in (sometimes publicly available) databases. These are important as most pedagogical systems are designed for novices who only utilise a subset of a language – there is no point having a system that supports advanced features novices do not know – therefore educators need to know what mistakes novices are making and how often.

These results are important to this research as it analyses the frequency of CEMs. Some work in this thesis will serve to show that the control group of this study is comparable to those in previous studies and is important for generalisation. Appendix B contains the top errors as reported by the eleven studies discussed below.

Hristova, Misra, Rutter, & Mercuri (2003) compiled a list of common Java errors committed by students, combining data from sources including teaching assistants, students and professors from 58 institutions, and members of SIGCSE[5], yielding a list of 62 errors, reduced to 20 as some were deemed too advanced for their students to encounter, or because the editor they used was capable of identifying the error in a useful enough way. Of these 20, 16 are syntax errors. Four have a one-to-one mapping with CEMs, and 12 do not (Altadmri & Brown, 2015). This complicated mapping is significant as it makes it difficult to relate a particular CEM to the error which causes it, and makes enhancing these CEMs difficult, as the offending code must be analysed in context to determine a helpful enhancement.

Thompson (2004) used an IDE called Gild which included ECEMs for 51 CEMs with ten errors accounting for 68% of all errors. Many of these I would not consider to be

[5] The ACM Special Interest Group on Computer Science Education (www.sigcse.org).

(absolute) novice errors, for instance those relating to abstract classes, interfaces and other ‘intermediate’ Java topics.

Jackson, Cobb and Carver (2005) used an improved version of Gauntlet – Flowers, Carver and Jackson’s IDE (2004), to identify the top 20 errors made by their students during a combined 20 semesters of work, making up 62.5% of all errors in their study.

Toomey (n.d.) combined the errors from Hristova, Misra, Rutter, & Mercuri (2003) and Jackson, Cobb, & Carver (2005), and added to these errors identified from 10 years of saved student submissions.

Jadud (2006) compiled a list of the ten most frequent errors from a dataset of about 70,000, representing 71% of all errors collected.

Dy and Rodrogo (2010) explored ‘non-literal errors’ which occur when the CEM does not match the actual error committed by the student. In doing so they compiled a list of the most common CEMs committed in their data. ‘Similar’ errors were grouped together, introducing difficulty when comparing these results to others.

Chan Mow (2012) gathered the most common errors generated by students and categorised them in the same way as (Hristova, Misra, Rutter, & Mercuri, 2003). The top eight of these represented 85% of all errors in his study.

Denny, Luxton-Reilly and Tabano (2012) confirmed the results of earlier studies, showing that a small number of syntax errors are encountered most often. They also provided evidence that students spend a large proportion of their time correcting these most common errors, and that high-performing students spend just as much time correcting certain errors as any other students.

McCall and Kölling (2014) attempted to improve on prior studies by investigating not the CEMs, but the logic errors (in code) that lead to them. They claim that the errors uncovered in their work are more precise, detailed, and accurate than a list of automatically generated CEMs. They accomplished this through a validated categorisation and the manual analysis of offending errors ‘by independent researchers of good reliability’ (p. 2589). The resulting list of error frequencies shows that prior work tended to group some distinct errors together when they should not be, allowing their errors to be listed more accurately. This may be a signal that the days of automatically compiling lists of errors is ending as this practice has provided all that it

can. In fact the authors note that their work ‘forms the basis of future work to improve compiler messages’ (p. 2859).

Brown, Kölling, McCall, and Utting (2014) collected errors through the BlueJ editor, which has at its core the ‘objects first’ philosophy of Java programming (Barnes & Kölling, 2011). This philosophy fundamentally alters the way that students write code and therefore the errors they generate. Thus, the authors concede that all data collected is a biased subset of the ways in which Java is taught. Although very popular the objects first approach has been a topic of fierce debate, a summary of which is available from (Bruce, 2004), and many teachers are convinced that this approach poses more problems than it solves (Beaubouef & Mason, 2005). Nonetheless, this work is a major step forward in that it describes a project which is continuously collecting error messages, through the BlueJ editor, from over 100,000 users, and this information is available to the research community for analysis. This database is clearly of more varied and reliable use than distilled lists (and often partial lists) presented in previous works. An initial top-ten errors representing 65.8% of all errors gathered through December 1, 2013 is presented as part of Appendix B. This is the most recent and by far most comprehensive study, utilising over 5,000,000 errors. However as discussed, the fact that these errors are only collected from the BlueJ editor introduces some bias.

It is apparent that when generating lists of common errors, many factors – most notably methodological or design decisions such as categorisation, what is relevant to the students in question, etc. – influence the final lists considerably. In addition, some studies combine errors by design, such as (Dy & Rodrigo, 2010). Particularly with this in mind, one needs not be an expert nor spend too long looking at Appendix B, to agree with McCall and Kölling’s conclusion: ‘There still is no real agreement about the most common problems students encounter’ (2014, p. 2589). In addition, Brown and Altadmri (2014) found that educators only have a weak consensus about the frequency of student errors. Finally, it is important to note that Java CEMs are not guaranteed to be stable across Java versions (Brown, Kölling, McCall, & Utting, 2014). The timeframe spanning the studies presented in Appendix B span 11 years and at least four Java versions, explaining more of the variation. Brown et al. acknowledged this, noting that they encountered several new messages that are partial reclassifications of previous errors (from older Java versions). They hope to address this issue in future work, by

producing stable classifications of errors that are independent of compiler error messages (p. 226).

2.3 Compiler Error Enhancement

Systems which enhance CEMs are predominantly used for teaching programming and specifically in systems designed for teaching novices. Eclipse[6] is the only industrial IDE which does so, and is discussed in Section 2.3.2. There is also one system which is an Eclipse plug-in, but as it was designed for novices, it is presented in Section 2.3.1.

2.3.1 Systems for Teaching Novices

This section focusses on systems using Java as the teaching language for several primary reasons: Java is currently the most popular teaching language; Decaf, which was developed in this study, is for Java; and most work in compiler error message enhancement since 2000 has been in Java. For references on languages other than Java, please see Chapter 1 and section 2.1. Before starting with Java however, one notable paper will be discussed first which dealt with the Pascal language.

Schorsch (1995) introduced CAP (Code Analyzer for Pascal), an automated tool to check Pascal programs for syntax, logic and style errors. CAP provided more user-friendly messages than the standard Pascal compiler. These messages were designed to inform the student what was wrong, why, and how to fix the problem. Such messages often included sample code as an example, and did not shy away from personal touches such as humour, which has also been used by at least one other system described later (Flowers, Carver, & Jackson, 2004). Scorsch claimed: “The numerous diagnostic messages reported by CAP empowers students by enabling them to find and fix errors that they previously could not find nor fix” (p. 169).

Java was introduced in 1995 and quickly saw uptake as a teaching language for CS1 courses. Soon after, efforts addressing the difficult Java compiler error messages began to appear, and continue to present day. In 1998 Hadjerrouit noted that “novice students encountered problems with the debugging and compilation tools of Java” (1998, p. 45). Also in 1998 Motil & Epstein developed JJ, a subset of Java which allowed students to focus on the programming concept, while not worrying about syntax. However, due to

[6] www.eclipse.org

lack of adoption, the authors shifted focus to trying to provide better compiler error messages, and programming over the web (Kelleher & Pausch, 2005). By 2000, JJ had "much emphasis placed on meaningful error messages, saying what the compiler thinks went wrong rather than what the compiler expected" (Farragher & Dobson, 2000, p. 7). Unfortunately most sources of information on JJ such as the project website itself are no longer available.

In 2002, Lang introduced a suite of tools to help students with Java. A core function of these was to produce better quality error messages. He noted that "The javac compiler does not always produce error message suitable for learners" (p. 95) and that the weakness of error messages produced by javac are a potential weakness of the Java development kit. The suite was in use from 1999 and was designed to help students become more self-sufficient, and to rely less on teacher intervention. Unfortunately no results on this were published.

Hristova, Misra Mercuri & Rutter (2003) introduced Expresso, a pre-compiler which scans programs for 20 common errors. These errors were identified through a survey of local students and faculty as well as faculty at other institutions. Expresso provides users with explanatory messages provided the error in question is one of those identified by Expresso. Although the authors describe their system as an "interactive tool that would do a better job generating error messages than existing compilers and also provide suggestions on how to fix the code" (p. 155), an assessment of the tool was left to future work. A drawback of Expresso is that error messages may not appear in line-number sequence due to the multiple-pass design. Being presented with errors which are not in line-number sequence is not desirable for at least two reasons. First, novice students often think sequentially – that is line-by-line. Second, students are often taught to tackle the first error message, due to the possibility of *cascading errors* (Burgess, 1999). These are not true errors, and are immediately resolved when the original error is resolved. To avoid being confused by cascading errors, Ben-Ari advises: "Do not invest any effort in trying to fix multiple error messages! Concentrate on fixing the first error and then recompile." (2007, p. 6). Following this line of thought, the inclusion of the second and subsequent 'errors' is a likely source of confusion and frustration, particularly for novices. This consideration has influenced the design of Decaf, as discussed in Appendix C.

In 2004, Thompson focussed on an Eclipse plug-in called Gild, specifically for novice Java programmers (2004). Gild was updated to include a feature with "extra error support" (p. 19) which consisted of error explanations and possible causes in plain English. Errors that Thompson felt required a longer explanation were explained in a wiki. Thompson noted that novices were unlikely to encounter all of the 347 errors that the Eclipse Java compiler provided at the time, so using several online resources Thompson generated a list of 51 errors for which Gild would provide ECEMs. At the end of the study, 10 of these errors accounted for 68% of all errors. This work, and the Gild editor, had many objectives, with the effects of compiler error enhancement making up three of six research questions. In addition, it was exploratory work with a small number of students – less than ten for the quantitative results, depending on the sub-study in question. The results were not conclusive as to whether or not students became faster at fixing their errors over the course of the study (one semester), or if their errors changed over the course of the semester. 57% (of 28 students) found Gild's extra error help feature "useful at least some of the time" (2004, p. 97). It was concluded that Gild needed more specific error messages and better coverage of errors most encountered by students.

Flowers, Carver & Jackson (2004) introduced a tool called Gauntlet which provided ECEMs. After targeting the top fifty beginner errors, the authors settled on nine which they believed to be most common. The authors used Gauntlet for 18 months in a first-year module which included programming. The authors posited that the quality of student work increased, time was saved, and instructor workload was reduced. However no empirical results were presented.

Coull (2008) introduced a framework for support tools that addresses both program and problem formulation for novices. One of the requirements of such tools is to present both standard compiler and enhanced support concurrently. Only three systems categorised by Coull met this requirement: CAP discussed earlier (Schorsch, 1995), one system that only focussed on logic (not syntax) errors (Etheridge, 2004), and another that focussed on the Verilog hardware description language (Moore, 2005). Coull also developed SNOOPIE using the framework, for learning Java programming. Although the scope of SNOOPIE was well beyond ECEMs, one of the primary focuses of the tool was just that. It was shown that this support was beneficial to a small group of students, particularly for non-trivial syntactic errors.

Hartmann, MacDougall, Brandt and Klemmer (2010) developed HelpMeOut, a "social recommender system that aids the debugging of error messages by suggesting solutions that peers have applied in the past" (p. 1019). However there was no work carried out to determine how helpful the system is to students.

Toomey (n.d.) developed Arjen, an extension to BlueJ (Toomey & Gjengset, 2011). Arjen identifies 29 errors and returns descriptive error information and advice. The system was piloted over 13 weeks with 25 students. During this the tool was only used sporadically. An end-of-semester evaluation revealed that students felt that Arjen was beneficial, but no rigorous quantitative or qualitative analysis was carried out.

All of the studies discussed so far put most focus on addressing the problem (providing ECEMs), but lack empiricism in determining if they make any difference to novices. Denny, Luxton-Reilly & Carpenter (2014) implemented an enhanced feedback system to users of CodeWrite (Denny, Luxton-Reilly, Tempero, & Hendrickx, 2011), a web-based tool designed to help students complete Java exercises. This 2014 study was the first recent work on the effect of Java ECEMs with control and intervention groups. It targeted 43 errors, organised into 9 categories. The feedback itself contains:

1. the line of code that contains the error,
2. a detailed explanation of what is most likely causing the error, and
3. a table containing:
    a. A code fragment including an error of the same category of the error that has been recognised,
    b. the same fragment with the error corrected and the differences highlighted, and
    c. an explanation of the error in the first fragment, and how it has been corrected in the second.

The system was used with students attempting exercises which required them to complete the body of a method for which the header was provided. Thus students were not writing code from scratch, and may not have been experiencing the full gamut of CEMs that novices may encounter. Students participated for a period of two weeks as part of an accelerated summer course. Students were required to complete ten lab exercises using the tool with 83 submitting at least one. To evaluate the enhanced feedback, the authors investigated the impact on:

1. The number of consecutive non-compiling submissions made while attempting a given exercise,
2. the total number of non-compiling submissions across all exercises, and
3. the number of attempts needed to resolve the most common kinds of errors.

Their analysis concluded that, with reference to the points identified above,

1. There were no significant differences between groups.
2. Although students viewing the enhanced error messages made fewer non-compiling submissions overall, the variance of both groups was high, and the difference between the means was not significant.
3. There was no evidence that the enhanced feedback affected the average number of compiles needed to resolve any of the common syntax errors.

The authors identify several possible reasons for their null results, including:

1. The majority of errors may have been simple enough to solve without the ECEMs.
2. Students in the intervention group may not have paid much attention to the additional information in the ECEMs, consistent with the results of Kummerfield and Kay (2003), Schorsch (1995) and Falkner (2014).
3. The ECEMs did not provide examples and explanations that students could relate to their own code.

The authors also note a threat to the validity of their findings – the raw compiler feedback shows up to two CEMs, while the enhanced feedback module displays only one in an attempt to reduce the complexity for students. This may allow some students to correct two errors at once while using the raw compiler messages, or may confuse other students by presenting more than one error to correct. Finally, they note that further study of ECEMs would be valuable not just in terms of their work but also that of others.

Although this work was published a year after work on Decaf began, it informed this research by providing:

1. A methodology for comparison
2. Results for comparison

3. Further justification, especially when taking into account the discussions it precipitated

2.3.2 Industrial Systems

Only one ‘professional’ or ‘industrial’ system (for Java) is mentioned with any consistency in the literature – the Eclipse development environment which comes with its own Java compiler and therefore its own CEMs. While Eclipse is not aimed at novices, Ben-Ari (2007) notes that Eclipse has much to recommend it, even in introductory courses, particularly for its “superior error messages” (p. 5) and support for identifying and correcting syntax errors, although these error messages are sometimes deemed confusing (Dy & Rodrigo, 2010). Ben-Ari’s recommendation is echoed by Debuse, and Lawley (2012), as part of a combination of tools for novices. Nonetheless, Eclipse is often deemed too advanced for beginners, its size and complexity outweighing any benefits, and has been the subject of efforts to improve it specifically for novices, including how errors are displayed (Reis & Cartwright, 2003; Storey, et al., 2003).

2.4 The Current Challenge

Despite the existing work, not enough has been done to make dealing with CEMs more effective for novices. CEMs are frequently inadequate (Denny, Luxton-Reilly, & Carpenter, 2014), and unfortunately words said in 2001 are still largely true: “As a Java programming instructor, you see a lot of strange errors. Eventually, you start to recognize certain patterns or favourite ways that novice programmers get mixed up” (Ziring, 2001, p. 1). According to Traver (2010), the lack of attention bad CEMs receive is attributed to an apparent feeling that programmers should adapt to compilers, not vice-versa, providing words which sum up the situation well (p. 1):

> *Error messages shown by compilers are, more often than not, difficult to interpret, resolve, and prevent in the future. The lack of computer support in this sense is somehow paradoxical. For instance, tools exist to help an analyst draw class diagrams; in some cases, these analysis tools even generate a basic code skeleton automatically. But, curiously, the difficulties faced by programmers, particularly those concerning compiler error messages, have not yet been addressed by mainstream compiler writers and remain a topic within the academic context.*

Finally, very recently rooted justification for this research is garnered from discussion surrounding the work by Denny, Luxton-Reilly, & Carpenter (2014) which occurred halfway through this study (Guzdial, 2014).

The next chapter lays out the research methodology and methods used in this study.

Chapter 3 Research Methodology and Methods

"*Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?*"

– Brian W. Kernighan & P.J. Plauger, The Elements of Programming Style, 1978 p. 10.

3.1 Theoretical Perspective

Ben-Ari noted in 1998 that there was a large literature on the psychology of computer programming (CP) and many of these researchers clearly held constructivist views, but there was a lack of literature on constructivism in computer science education (CSE) (1998). Ben-Ari assessed the relevance of constructivism for CSE and supplied a theoretical basis for its application. Odekirk-Hash and Zachary (2001) solidified this, building an online tutoring system to see if an automated system could effectively guide programming students through a constructivist learning session organised around Vygotsky's scaffolding. Several other authors continued along this trajectory early in this decade. Research on CP in general has more recently been studied from a Piagetian perspective (Kramer, 2007), and neo-Piagetian perspectives (Lister, 2011; Corney, Teague, Ahadi, & Lister, 2012), However these studies were focussed on the behaviours of professional software engineers, not novices.

The theoretical perspective of this research based on the deterministic nature of CEMs is post-positivist (Phillips & Burbules, 2000). Ryan (2006) provided the following characteristics of post-positivist research:

- Research is broad rather than specialised – lots of different things qualify as research;
- Theory and practice cannot be kept separate. We cannot afford to ignore theory for the sake of 'just the facts';
- The researcher's motivations for and commitment to research are central and crucial to the enterprise;

- The idea that research is concerned only with correct techniques for collecting and categorising information is now inadequate.

Compilation can be considered perfectly deterministic, as a given compiler will always provide the same output (CEMs). Nonetheless, the sometimes not one-to-one mapping of student errors to CEMs discussed in Section 2.2 is a complexity. Also rooted in a post-positivist perspective is the use of intervention and control groups, each providing a large quantitative dataset, and the fact that within each group, each programming session can form a 'case-based' problem session (Robins, Rountree, & Rountree, 2003).

Although efforts are taken to keep myself as external to the research as possible, I am aware that I am the principal designer of the Decaf software and the study itself, and that this could have the potential to affect results – particularly in the introduction of limitations and biases, clearly distancing my post-positivist perspective from that of purely positivist (Check & Schutt, 2011). It is here that in terms of epistemology, objectivity is important – having the researcher manipulate and observe in a dispassionate, objective manner (Mertens, 2014).

As discussed further in the following section, triangulation – more than one source of data and more than one method of data collection – will be used, as it is a post-positivist perspective that as all measurement and observation is fallible, making multiple measurements and observations is important (Gray, 2013; Trochim, 2006). For my post-positivist ethical stance, see Section 3.4.

3.2 Research Methodology

The research methodology is a single instrumental case study, where a particular case is examined to provide insight into an issue or refinement of theory (Stake, 1998) with an explicit expectation that what is learned can be used to generalise to other cases. This methodology is compatible with a mix of qualitative and quantitative methods as proposed. This case is bounded by the CS1 module, my institution, time (2013-2014), and two cohorts of students. Note that the fact that this research only considers the Java programming language is considered a limitation, discussed in Section 4.5, and not explicitly a boundary of the case-study methodology. The fact that the CS1 module (which at my institution uses Java) is a case-study boundary is the reason that this research only considers Java.

There were two cohorts involved in his study, one intervention (Autumn 2013) and one control (Autumn 2014), each of approximately 100 students. The control/intervention approach is standard in quantitative research to address reliability and validity, particularly when combined with triangulation (Zuber-Skerritt, 1992). The groups were as identical to one another as possible in environment and very similar in profile. The only planned difference was that the control group used Decaf in 'pass through' mode, where there is no enhancement of CEMs. The intervention group used Decaf in 'normal' mode, with the ECEMs presented alongside the CEMs. After the six week study period, the control group was given access to the intervention version of Decaf. Both groups were free to continue using Decaf for the duration of the module, although they were also provided with another compiler at the six-week point. Figure 3.1 shows a timeline view of each group.

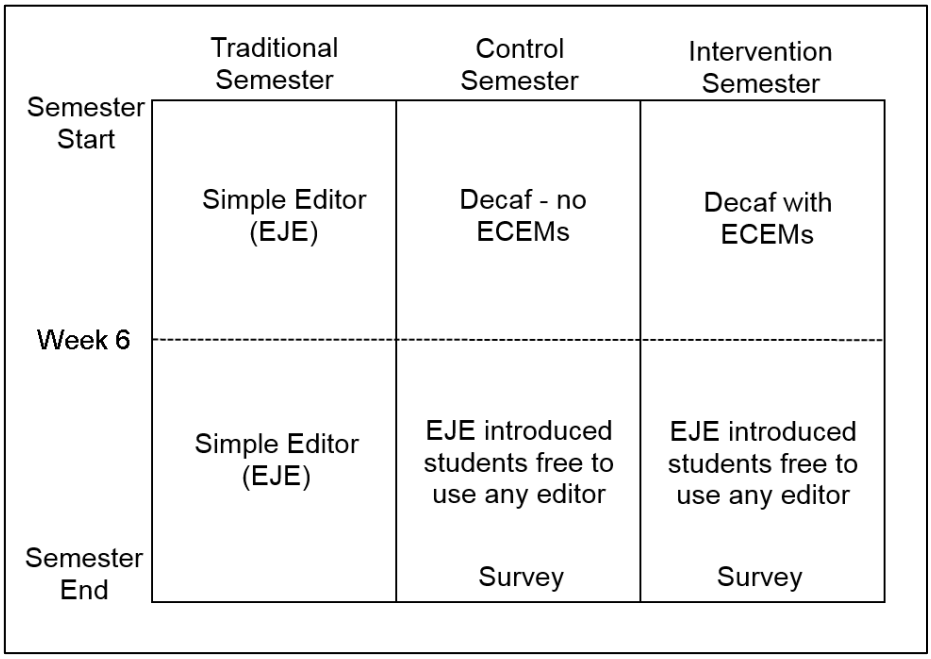

Figure 3.1 Overview of traditional (pre-2013), intervention (2013), and control (2014) semesters, highlighting what compilers were available to students.

The research environment was the College of Computing Technology in Dublin Ireland. The CS1 module was "Computer Programming 1", a five credit[7] module that is designed to teach programming with no prior knowledge. The lecturer was aware that the research is being conducted and also provided objective comments on the software

[7] European credit transfer and accumulation system credits

design as well as providing some of the technical (data-logging) development, not on designing ECEMs.

The students are enrolled in a three year BSc in Information Technology[8]. They are approximately 85% international and 15% local (Irish). Table 3.1 presents a profile of each group.

Table 3.1 Profiles of control and intervention groups.

Group	Average attendance	Average age	% female/male
Control (2014)	97	26	28/72
Intervention (2013)	95	28	25/75

Efforts were made for module delivery to be as equivalent for each group as possible including schedule, content and assessment. Students completed weekly lab assignments aligned to weekly topics summarised in Table 3.2.

Table 3.2 Summary of weekly topics for control and intervention groups.

Week 1	Week 2	Week 3	Week 5	Week 5	Week 6
Data types, basic mathematics	Decision making, logical & relational operators	Loops	Arrays	User input, exceptions, type casting/parsing	Switch and mid-term revision

3.3 Research Methods

Although the primary focus of this study was quantitative, it also had qualitative aspects, with multiple sources of data and methods of data collection, described below. Table 3.3 shows how each research question is aligned with these methods.

Table 3.3 Alignment of research questions with methods.

Question	Logged data	Survey	Interviews
Do enhanced compiler error messages reduce the overall number of errors?	X		

[8] Irish National Framework of Qualifications level 7, Quality and Qualifications Ireland validated. (www.qqi.ie)

Do enhanced compiler error messages reduce the number of errors per student? If so, for what compiler error messages in particular?	X		
What effect do enhanced compiler error messages have on students, particularly struggling students?	X	X	X
Do enhanced compiler error messages reduce the number of repeated errors?	X		
What are learner and educator views on enhancing compiler error messages?		X	X

3.3.1 Quantitative

3.3.1.1 Logging

Each group had the following data logged, for each CEM generated:

- Compiler ID (anonymous)
- Line of code and class generating CEM
- CEM
- ECEM (for intervention group)
- Date / time

Data logging was known to students, but transparent. Shapiro-Wilk testing indicated that most data did not have normal distributions, as expected for some of it, such as the distributions of errors (see Section 4.1), and non-parametric Mann-Whitney *U* tests were utilised. Where Shapiro-Wilk tests did show that data was normal, a Student's *t*-test was also used. For paired data, a Wilcoxon signed-rank test was utilised. All results are two-tail and the significance level, $\alpha = 0.05$. All tests are robust for the sample sizes involved.

3.3.1.2 Survey

To get an idea of how learners viewed and experienced compiler error enhancement with Decaf a short optional and anonymous survey was employed. Both control and intervention groups completed the survey in December, at the end of the semester that they began with Decaf, but approximately six weeks after the end of the study period.[9] The control group had access to regular Decaf (which does enhance CEMs, which the intervention group used during the study period) after the study period. Both groups

[9] Before assessment of the modules was complete.

also had access to at least one other more advanced IDE. This allowed both groups to try other IDEs and to reflect on Decaf with the experience of at least one other environment. The intervention group was surveyed before September 2014 when this thesis formally began, but all analysis took place after.

The survey was comprised of four mandatory Likert questions (providing quantitative data), each with an optional open-ended (providing qualitative data) field asking "Please explain (optional)". The questions were designed with some influence from Rigby and Thompson (Rigby & Thompson, 2003).

As chapter 4 presents data on students starting with the errors they make, I present survey findings in Section 4.4.1 presenting a complete picture of the student experience.

3.3.2 Qualitative

3.3.2.1 Surveys

In addition to the open-ended parts of the Likert questions discussed in Subsection 3.3.1.2, the survey also included four (stand-alone) open-ended questions to illicit deeper feedback on Decaf and the student experience of using it.

3.3.2.2 Interview

The lecturer was interviewed once for each group to gain his perspective on the student experience providing a second source of qualitative data. The lecturer was the same for both intervention and control groups, and was interviewed by myself twice, once for each group, to gain his perspective on the student experience with Decaf. Following my post-positive theoretical perspective I kept myself distanced from the experiment itself and this provided me an important insight into how the students and the lecturer used Decaf.

The lecturer is familiar with the module (years ago he took it himself), and is comfortable with the environment. As he played a supporting role in this research he was familiar with the study itself and was intimately familiar with the Decaf software. It is important to note that during lab sessions the lecturer often sat side-by-side with students using Decaf in both groups, giving him a unique perspective in its use by both groups of students under his supervision.

The interviews were semi-structured, recorded (later transcribed) and followed the same questions and format and the same interview guide used. The first, for the

intervention group was held in November 2014 and the second, for the control group was conducted in January 2015. The goal was to get the educator's point of view in rich, detailed explanations on the following points:

a. Lecturer experience
    i. Methodology
    ii. How did it differ from the 'norm'
b. Student experience
    i. How did it affect student behaviour (compilation)?
    ii. How did students interact with Decaf? Positively/Negatively?
c. Improvements
    i. Software
    ii. Use

Results, findings and analysis are found in Section 4.4.2.

3.4 Ethical Considerations

Ethical considerations aligned with the post-positivist perspective: privacy should be respected; consent should be sought and informed, and beneficence should be an objective (Mertens, 2014). I identified the following issues:

1. Survey:
    a. Confidentiality must be preserved
    b. Data storage must be secure, including
    c. Data should be destroyed as soon as possible
    d. Conflict of interest was possible, should I be too close to the research
2. The Decaf software, specifically the automatic logging of student data:
    a. Requirement for an informed opt-in policy
        i. Built-in to Decaf
    b. Requirement for student awareness of what is being logged and how
        i. Consent and information forms
3. It was not known if either group was at a disadvantage relative to the other. In this light, research and data collection was conducted with minimum impact in minimal time and the control group was given full access to the intervention software at the end of the study period.
4. Ethical clearance was obtained from the College of Computing Technology, which was considered sufficient by the Ethics Committee of the Dublin Institute of Technology.

3.5 Technical Implementation Details

Decaf is written in Java and uses the Swing API for its graphical user interface. It runs on Windows and Apple OS X. It uses the Runtime class to interface with the local environment and directly invokes javac of the local JDK. When javac returns a CEM to Decaf, the CEM and student code are inspected and programs logic attempts to determine the cause of the error which caused the CEM. If this is achieved, an ECEM is presented to the user *alongside* the original CEM. Figure 3.1 shows a schematic of how Decaf interacts with the user, the JDK/javac, and the logging database.

The CEMs which are checked were largely compiled from findings detailed in Appendix B. Details of some individual CEMs including likely causes from (Ben-Ari, 2007) were used. In addition I included a few basic and common errors which I have seen occur with beginners (but not appearing in Appendix B):

- class <class name> is public, should be declared in a file named <class name>.java
- '.' expected
- illegal character '<character>'
- reached end of file while parsing
- unclosed character literal
- unreachable statement
- array required, but <type> found

Table 3.4 shows all CEMs enhanced by Decaf. These were categorised so that data could be analysed by category if required. In cases where a CEM has multiple causes (lacks a one-to-one mapping with errors), further program logic attempts to determine the specific cause of the error by analysing the offending line of user code. One such example is the CEM *cannot find symbol*. Ben-Ari (2007) notes that this error can be caused by inconsistencies between the declaration of an identifier and its use, a non-exhaustive list of syntax errors resulting in such is:

a. misspelled identifier (including capital letters used incorrectly)
b. calling a constructor with an incorrect parameter signature
c. using an identifier outside its scope.

A brief code example demonstrating such a case (cause a.) was shown in Figure 1.1.

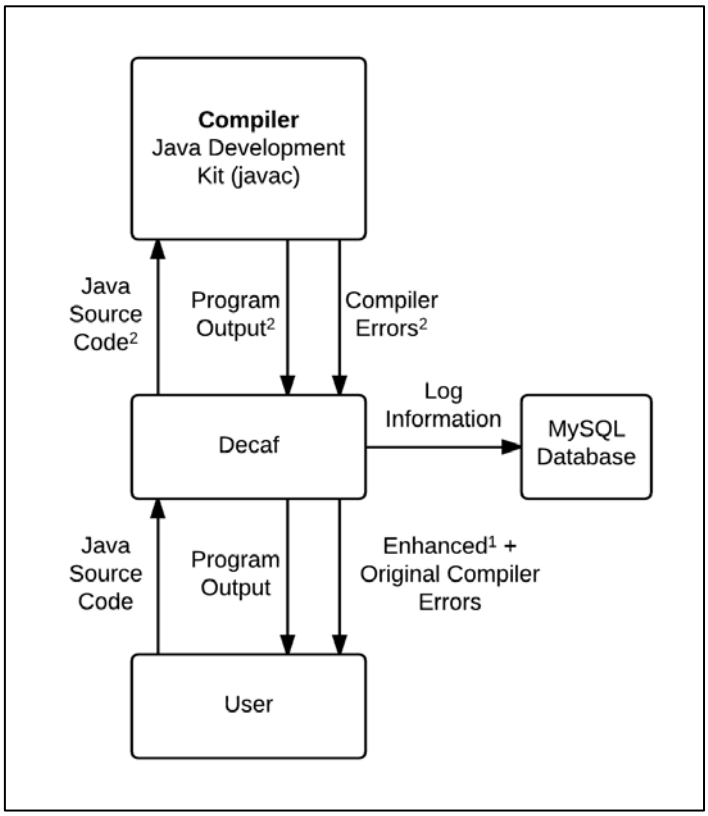

Figure 3.2 Schematic of Decaf and interactions with user, JDK/javac and database. 1 in 'pass-through' mode, the enhanced error is omitted. 2 through the runtime environment.

Table 3.4 CEMs which are enhanced by Decaf, and corresponding error categories.

Compiler Error Message	Category
bad operand type *type_name* for unary operator '*operator*'	Bad operand type
bad operand types for binary operator '*operator*'	Bad operand type
package <package_name> does not exist	Class/package errors
class <class_name> is public, should be declared in a file named <class_name>.java	Class/package errors
class, interface, or enum expected	Class/package errors
variable <variable_name> is already defined in method <method_name>	Incorrect assignment / creation of variable / object
variable <variable_name> might not have been initialized	Incorrect assignment / creation of variable / object
invalid method declaration; return type required	Incorrect return statements
missing return statement	Incorrect return statements
'.' expected	Missing/unexpected/illegal character/identifier/symbol/start of expression
';' expected	Missing/unexpected/illegal character/identifier/symbol/start of expression
<identifier> expected	Missing/unexpected/illegal character/identifier/symbol/start of expression
cannot find symbol	Missing/unexpected/illegal character/identifier/symbol/start of expression
illegal character: '<character>'	Missing/unexpected/illegal character/identifier/symbol/start of expression
illegal start of expression	Missing/unexpected/illegal character/identifier/symbol/start of expression
'(' expected	Misused or unmatched braces or parenthesis
'(' or '[' expected	Misused or unmatched braces or parenthesis
')' expected	Misused or unmatched braces or parenthesis
'[' expected	Misused or unmatched braces or parenthesis
']' expected	Misused or unmatched braces or parenthesis
'{' expected	Misused or unmatched braces or parenthesis
'}' expected	Misused or unmatched braces or parenthesis
reached end of file while parsing	Misused or unmatched braces or parenthesis
unclosed character literal	Unclosed literal
unclosed string literal	Unclosed literal
not a statement	Other
possible loss of precision	Other

unreachable statement	Other
array required, but <type> found	Type mismatches
incompatible types OR incompatible types: <type> cannot be converted to <type>	Type mismatches

Appendix C contains a discussion on the design of the Decaf software and screenshots showing the user's view. Appendix D includes a discussion on the design of the ESEMs. Appendix E contains information on categorising Decaf in the taxonomy of Kelleher and Pausch (2005).

The next chapter presents findings and analysis, organised around the research question and sub-questions.

Chapter 4 Findings and Analysis

> "…error on string tokenizer :/#%@ˆ expected, what the hell does that mean, honestly"
>
> - *CSC 115 student*, Gild questionnaire (Thompson, 2004, p. 1)

This chapter presents and analyses findings, using the sub-questions of the research question as guides. The overall aim of this research is to answer the question:

Do enhanced compiler error messages help students who are learning to program?

The sub-questions are:

1. *Do enhanced compiler error messages reduce the overall number of errors?*
2. *Do enhanced compiler error messages reduce the number of errors per student? If so, for what compiler error messages in particular?*
3. *What effect do enhanced compiler error messages have on students, particularly struggling students?*
4. *Do enhanced compiler error messages reduce the number of repeated errors?*
5. *What are learner and educator views on enhancing compiler error messages?*

It is worthwhile first looking at the errors generated by the control group, and comparing them to those from previous studies. Table 4.1 shows the ten most frequent CEMs recorded in the control group, representing 73% of all errors from that group. This shares six CEMs with the top ten from (Brown, Kölling, McCall, & Utting, 2014) and (Jackson, Cobb, & Carver, 2005), five with (Tabano, Rodrigo, & Jadud, 2011) and (Dy & Rodrigo, 2010), and four with (Jadud, 2006). As noted in section 2.2 however, different authors sometimes combine different CEMs, and report them in different ways (see notes below Table 4.1). In addition these studies span ten years, and most likely four Java versions[10]. Students in this study used Java SE 7. Although SE 8 was released in March 2014, SE 7 was the default version to download until October 26, 2014 (Weildt, 2014).

[10] For the most comprehensive history of Java versions (with dates) see: https://en.wikipedia.org/wiki/Java_version_history

The control group generated similar errors to five other studies spanning ten years, indicating that the control group in this study is comparable to those in others. Figure 4.1 shows that the distributions of the top ten errors from the control group of this study and the other five in Table 4.1 are very similar, providing further indication that the control group in this study is comparable to others.

Table 4.1 Top 10 errors from this study (control group) and five other Java studies.

Error Description	**% of all errors (control group)**	(Brown, Kölling, McCall, & Utting, 2014)	(Jackson, Cobb, & Carver, 2005)	(Tabano, Rodrigo, & Jadud, 2011)	(Dy & Rodrigo, 2010)	(Jadud, 2006)
cannot find symbol*	**16.0**	17.7**	14.6	~18**	18.9**	16.7**
‘)’ expected	**11.5**	6.5†	3.8	~10†	9.6†	10.3†
‘;’ expected	**10.7**	9.5	8.5	~12	11.7	10.0
not a statement	**7.4**	3.0	2.5			
illegal start of expression	**6.3**	4.4	5.7	~5	5.2	5.0
reached end of file while parsing	**4.9**					
illegal start of type	**4.6**					
‘else’ without ‘if’	**4.0**					
bad operand types for binary operator	**3.9**					
<identifier expected>	**3.8**	3.6	4.5	~9	3.7	
% Total	**73.0**	65.8	51.8	~69	79.9	71.9
Total errors	**28,860**	$> 5\times10^6$	559,419	24,151	~14,500	~70,000

*Some studies broke this CEM down into: unknown variable, unknown method, unknown class, and unknown symbol. As the students in this study had not yet studied methods or classes, it is reasonable to assume that most ‘cannot find symbol’ errors were actually “cannot find symbol – variable” errors. Manually looking at many of these errors in the data supports this.

** “Unknown variable” or “Cannot find symbol – variable” (See * above)

† “Bracket expected” or “‘(’ or ‘)’ or ‘[‘ or ‘]’ or ‘{‘ or ‘}’ expected”

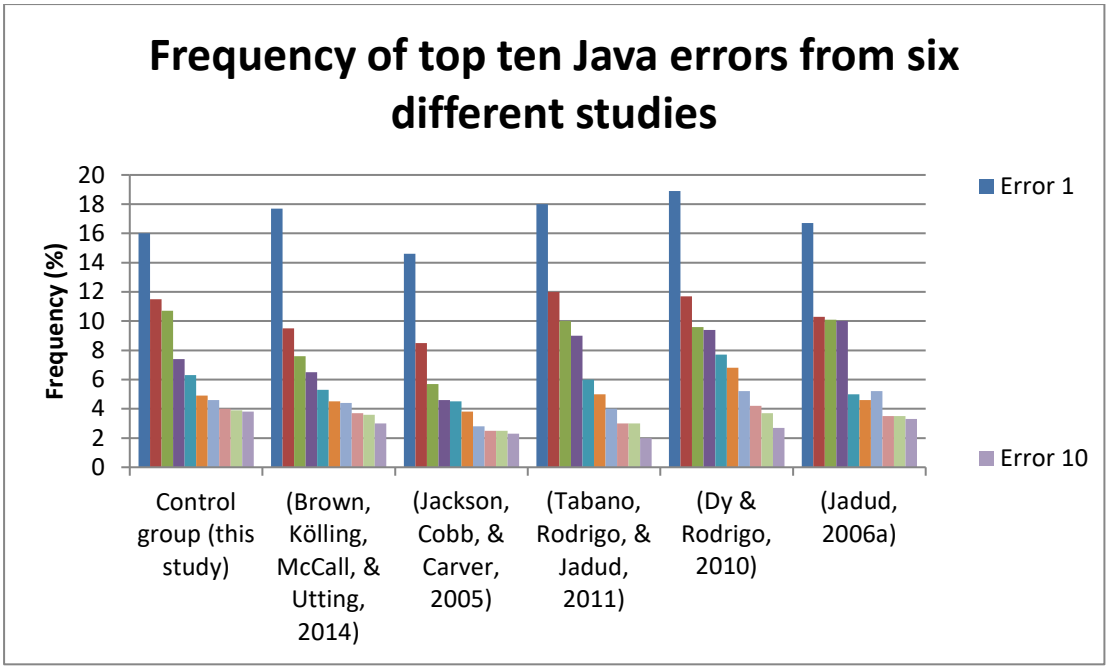

Figure 4.1 Frequency of the ten most frequent Java errors from this study (control group) and five others.

Jadud noticed that top Java errors he collected had a similar distribution to five other studies using languages other than Java (2006). Inspired by his analysis, Figure 4.2 shows the frequency of the nine most frequent errors from this study's control group against those from three languages Jadud reported on: Haskell (Heeren, Leijen, & van IJzendoorn, 2003); FORTRAN (Moulton & Muller, 1967); and COBOL (Litecky & Davis, 1976), as well as the most recent study on Java (Brown, Kölling, McCall, & Utting, 2014).

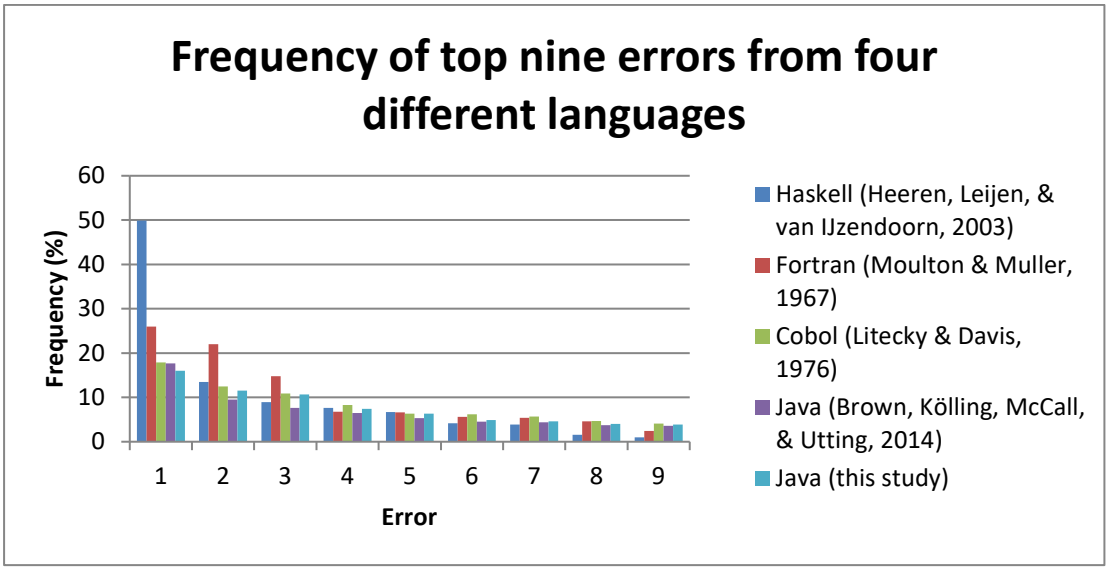

Figure 4.2 Frequency of nine errors from four different languages.

Each of the numbered errors are different as the languages are different (with the exception of some of the Java errors, see Table 4.1), and there is no way of evaluating why this distribution is common[11] across so many languages. However Jadud posits two possible reasons: the programmer and the grammar (2006). If indeed the reason is programmer behaviour, it would support the idea that the students in this study are not only similar to those in other studies involving Java, but involving many other languages as well. This would be important in generalising the results of this study. Similarly, if this commonality is due to the grammar of the languages, it could be taken as evidence that results for one language could potentially be generalised to others, with obvious complications involving systematically and reliably generalising errors in one language to another.

4.1 Do enhanced compiler error messages reduce the overall number of errors?

48,800 errors were recorded, representing 74 distinct CEMs, including all 30 for which Decaf provides ECEMs. The full list of CEMs is shown in Appendix F.

Table 4.2 shows the overall number of errors recorded, the number of compiler IDs, the average attendance and the number of students submitting lab work over the study period for both groups[12]. The intervention group logged 32% fewer errors overall, for a very similar number of students.

Table 4.2 Profiles of control and intervention groups.

Group	Number of errors	Number of compiler IDs	Average attendance	Average number of lab submissions
Control	29,015	122	97	90
Intervention	19,785	120	95	94
Total	48,800	242		

The number of compiler IDs is greater than the average attendance and number of students submitting lab work because the anonymous compiler ID is assigned per install of Decaf. If a student reinstalls Decaf, a new compiler ID is issued. This is discussed further in Section 4.5. Looking at the data I noticed a number of compiler IDs generating very few errors, most of these occurring in the first week of the study period.

[11] The most common error in Haskell is the most extreme outlier in this data. See (Jadud, 2006, p. 69) for a possible explanation. Removing this outlier makes the distributions even more similar.

[12] Although data was logged for a six-week period for each group, we only used data from weeks 2-5 for analysis, when each group was working at steady-state. During week 1 Decaf was being installed and during week 6 students were transitioning to another editor.

This is consistent with the lecturer noting that a few students did reinstall Decaf early on. Considering both groups, there was an average of 202 errors per compiler ID. Throughout this section when speaking of students, I am referring to a compiler ID, as this is the metric closest to the data and the students. Also, we will see soon that the number of compiler IDs in each group will be equivalent after filtering the data. The attendance and number of students submitting lab assignments are taken as corroborating evidence that the number of students in both groups is very similar, well within 5% regardless of metric.

Table 4.3 shows a summary after filtering Compiler IDs recording less than an average of ten errors per week from the data. This strikes a good balance between removing compiler IDs with very low activity and retaining those which are the result of a Decaf reinstall, but which generated a representative and useful amount of data. Other studies such as (Jadud, 2006), have filtered data similarly, for similar reasons.

Table 4.3 Group profiles filtered for inactive students.

Group	Number of errors	Number of compiler IDs	Average attendance	Average number of lab submissions
Control	28,861	108	97	90
Intervention	19,628	104	95	94
Total	48,489	212		

Figure 4.3 shows histograms of errors for both groups. A Wilcoxon signed-rank test (two-tail) showed that the number of errors was greater for the control group (*Mdn* = 34) than for the intervention group (*Mdn* = 18), Z = -4.29, p = 1.74e-05.[13] Both distributions look similar, and this is expected – we do not expect Decaf to fundamentally change the nature of errors other than the possibility of reducing them, and minor shifts in relative frequencies (See Section 4.1.1). Figure 4.4 shows a strong positive linear correlation in number of errors per CEM, as well as control group having more errors per CEM.

[13] The significance level α for all tests in this thesis is 0.05. *Mdn* is the median, Z is the test statistic. If $p < \alpha$ the result is be considered significant.

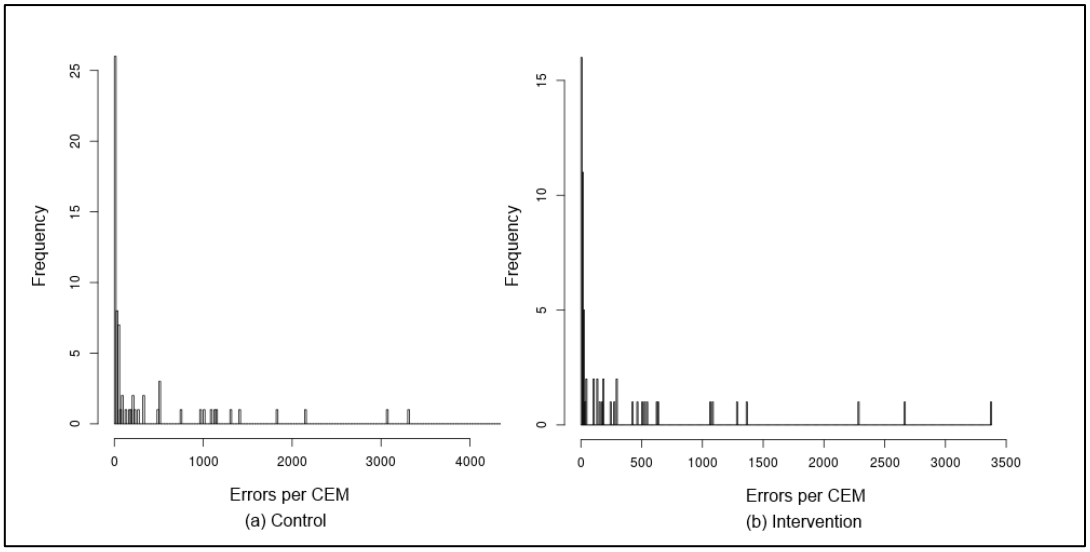

Figure 4.3 Histograms of errors per CEM for (a) control and (b) intervention groups. Note that the axes have different scales.

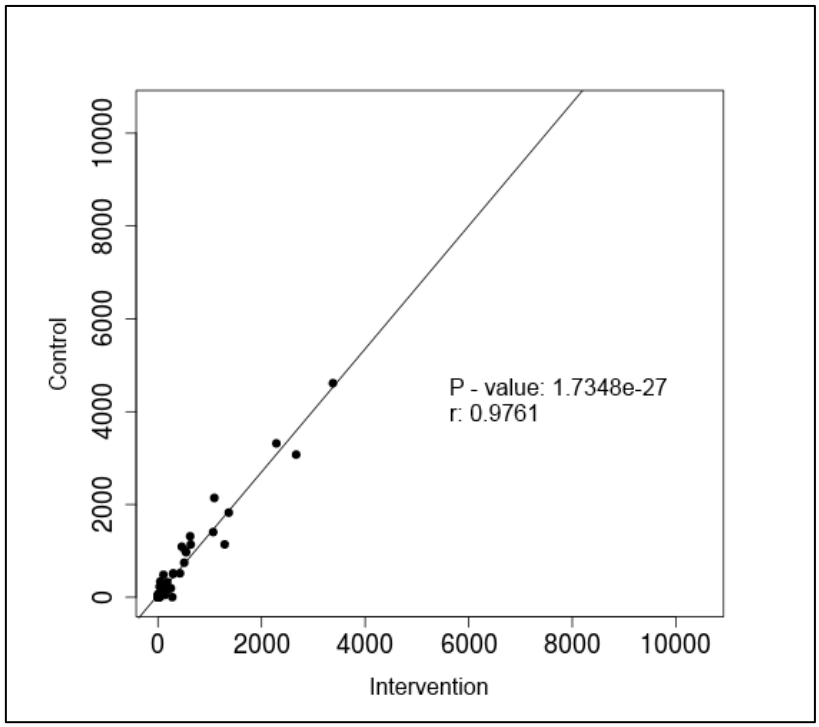

Figure 4.4 Correlation of errors per CEM for between the control and intervention groups.

4.1.1 High-Frequency Errors

Looking at the data it was obvious that most CEMs have a relatively low frequency. In fact for five CEMs, only one group logged errors. Noting this, and having removed very inactive compiler IDs, we next sought to simplify the data by filtering out CEMs with very low frequencies. As found in Section 4.1, the top few CEMs account for the vast majority of all error occurrences with the top 15 CEMs accounting for 86.3% of all

errors. The 16th most frequent accounts for only 1.6% of all errors, and subsequent CEMs even less. Filtering the data to keep the 15 most frequent errors resulted in eliminating eight compiler IDs altogether, accounting for a total of 154 errors (< 0.5%).

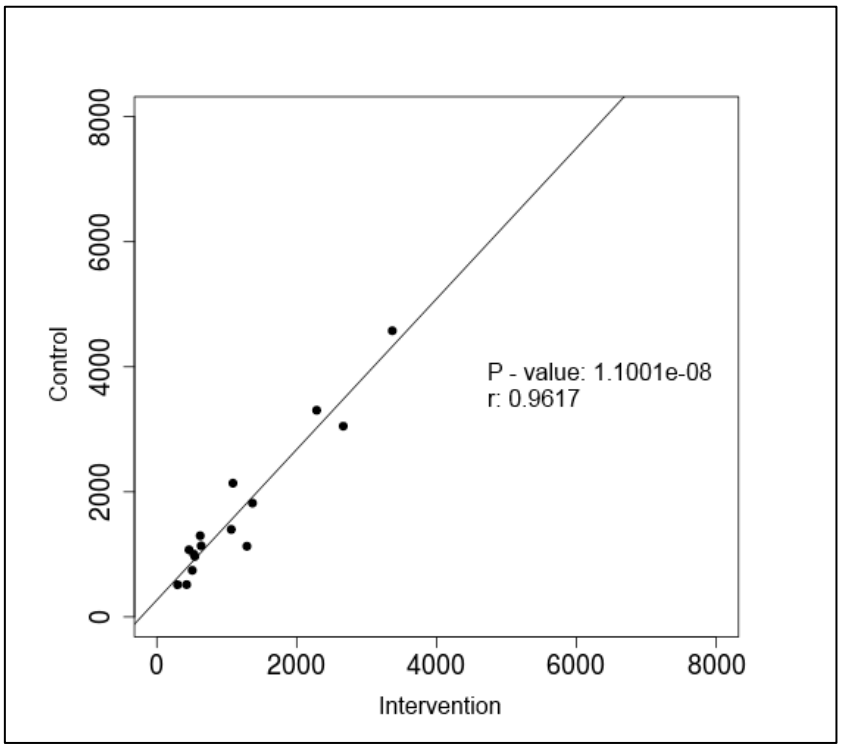

Figure 4.5 Correlation of errors per CEM between the control and intervention groups (15 most frequent CEMs).

Figure 4.5 shows that the strong linear correlation present in Figure 4.4 is preserved and that the control group had more errors per CEM than the intervention group. Coincidentally this made the number of compiler IDs equivalent across groups, as shown in Table 4.4.

Table 4.4 Group profiles filtered to eliminate inactive students and infrequent CEMs.

Group	Number of errors	Number of compiler IDs	Average attendance	Average number of lab submissions
Control	24,689	102	97	90
Intervention	17,144	102	95	94
Total	41,833	204		

Figure 4.6 shows histograms of errors per CEM after filtering data by inactive students and infrequent errors. A Wilcoxon signed-rank test (two-tail) showed that the number

of errors was greater for the control group (*Mdn* = 1,135) than for the intervention group (*Mdn* = 627), *Z* = -3.17, *p* = 0.002.

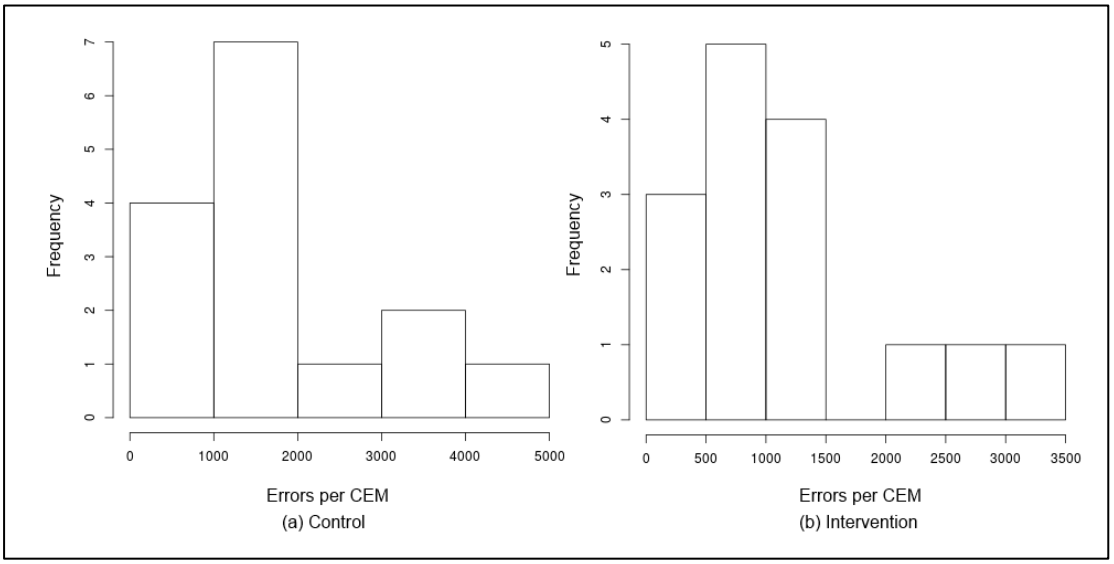

Figure 4.6 Histograms of errors per CEM (for the 15 most frequent CEMs) for (a) control and (b) intervention groups. Note that the axes have different scales.

Figure 4.7 shows the 15 most frequently encountered CEMs for both the control and intervention groups. There is a minor shift in rank for some CEMs: CEM 7 moves from 12% in control to 16% in intervention, CEM 39 from 5% to 8%, and CEM 74 moves down from 9% to 6%. Again, we do not expect Decaf to fundamentally change the nature of errors other than the possibility of reducing them, and minor shifts in relative frequencies – it is those shifts we see here.

Figure 4.8 shows the 15 most frequently encountered CEMs for both groups, representing 86.3% of all errors recorded. 12 of these CEMs are enhanced by Decaf, representing 74.7% of all errors. It can be seen that the number of errors is lower for the intervention group for all errors except CEM 39 which is *not* enhanced by Decaf (representing 5% of all errors), and with the smallest relative difference between groups of all top 15 CEMs (13.9%). The CEM with the greatest difference was 12, where the intervention frequency was 42.8% of control. Table 4.5 shows this data in tabular form, and we will refer to this and similar data again.

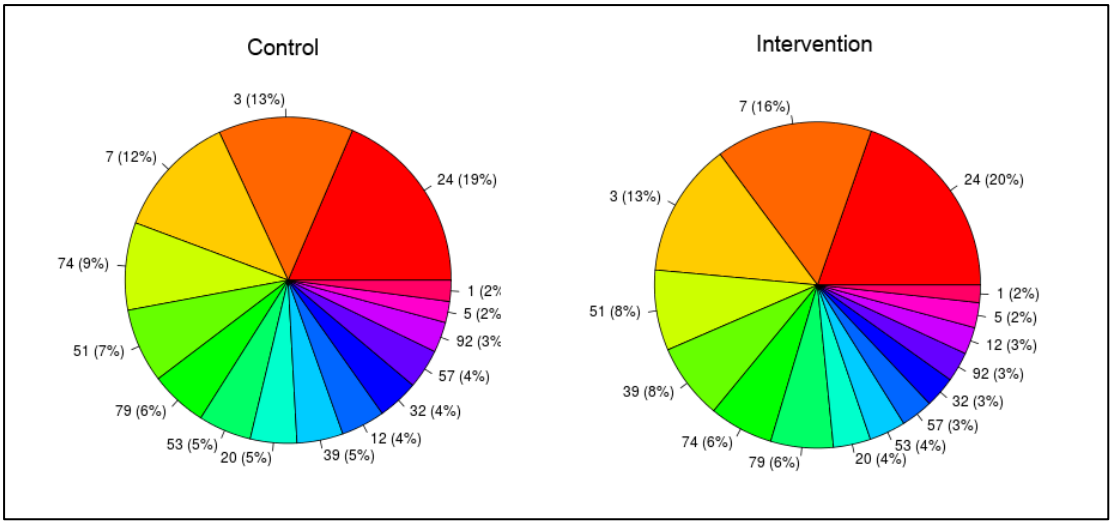

Figure 4.7 Frequency of top 15 CEMs for control and intervention groups. Numbers map to CEMs (see Figure 4.8). For instance, 24 is *cannot find symbol.*

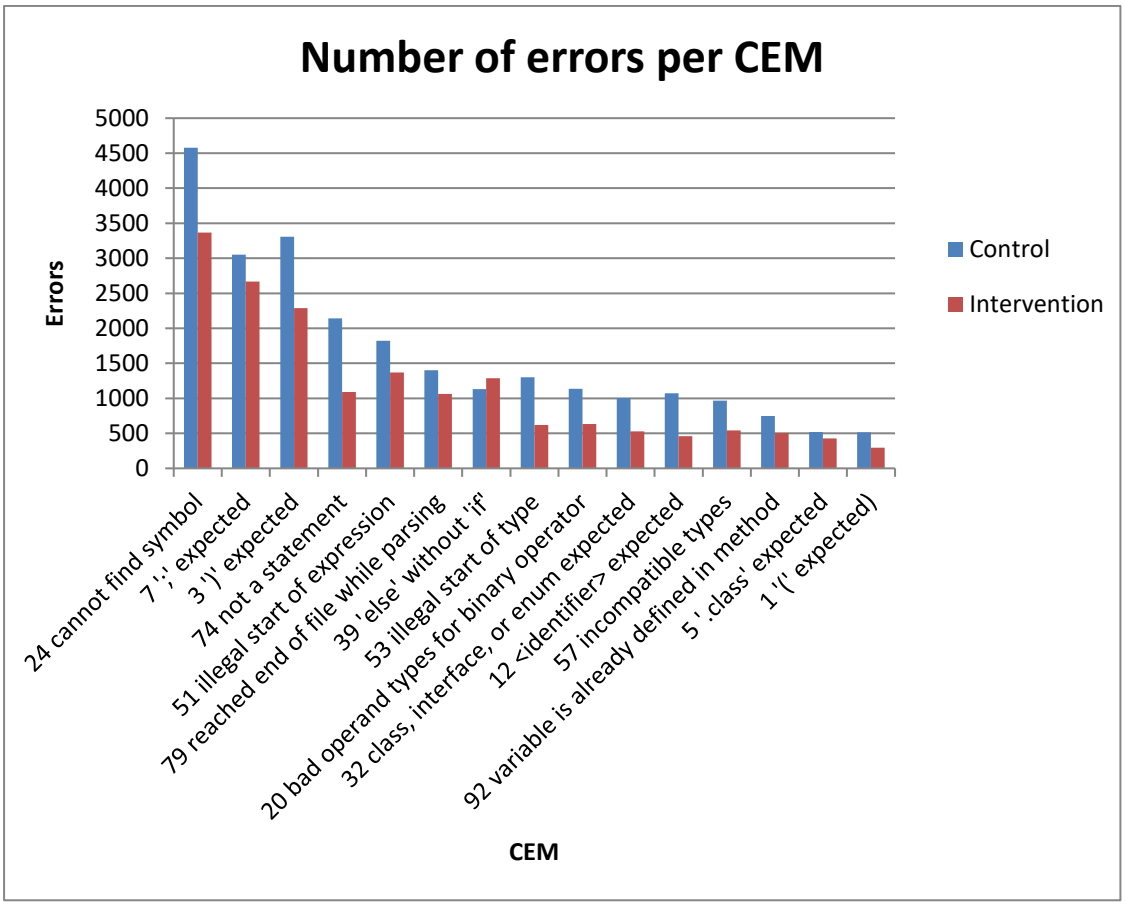

Figure 4.8 Number of errors per CEM (top 15 CEMs).

In the next section we will explore the number of errors per student and if these differences are significant in that context.

Table 4.5 Number of errors per CEM (top 15 CEMs).

Error	Description	Control errors	Intervention errors	Enhanced by Decaf?
24	cannot find symbol	4577	3368	Yes
7	';' expected	3051	2666	Yes
3	')' expected	3306	2286	Yes
74	not a statement	2140	1088	Yes
51	illegal start of expression	1821	1368	Yes
79	reached end of file while parsing	1400	1064	Yes
39	'else' without 'if'	1131	1288	No
53	illegal start of type	1299	620	No
20	bad operand types for binary operator '*operator*'	1138	634	Yes
32	class, interface, or enum expected	1005	529	Yes
12	<identifier> expected	1074	460	Yes
57	incompatible types OR incompatible types: *type* cannot be converted to *type*	969	543	Yes
92	variable *variable_name* is already defined in method *method_name*	746	507	Yes
5	'.class' expected	517	426	No
1	'(' expected	515	297	Yes

4.1.2 Summary

This subsection sought to answer the question: *Do enhanced compiler error messages reduce the overall number of errors?* The answer that emerged is yes, particularly for high-frequency errors. The extent to which this is generalizable to other students will be included in future work.

4.2 Do enhanced compiler error messages reduce the number of errors per student? If so, for what compiler error messages in particular?

Table 4.6 shows the average number of errors and average number of CEMs per student for each group.

Table 4.6 Group profiles showing average errors and CEMs per student.

	Average errors per student	Average CEMs per student
Control	265	20
Intervention	188	16
Overall	228	18

The average number of errors per student for the control group is 265, resulting in an average of 20 CEMs. The intervention group had an average of 188 errors per student, resulting in an average of 16 CEMs. Figure 4.9 shows histograms of the number of errors per student for both groups for all CEMs. It can be seen that a large number of students have relatively few errors compared to a small number of students who have relatively large number of errors. A Mann-Whitney *U* test (two-tail) did not show a significant difference in the number of errors per student, however a one-tail test did indicate that the number of errors was greater for the control group (*Mdn* = 169) than for the intervention group (*Mdn* = 135), *U* = 4,799, *p* = 0.043.[14] It is tempting to expect a lower number of errors per student for the intervention group given the findings thus far, so one and two-tail results are calculated here. To investigate further, we will now inspect the top 15 CEMs in the search for more definitive findings.

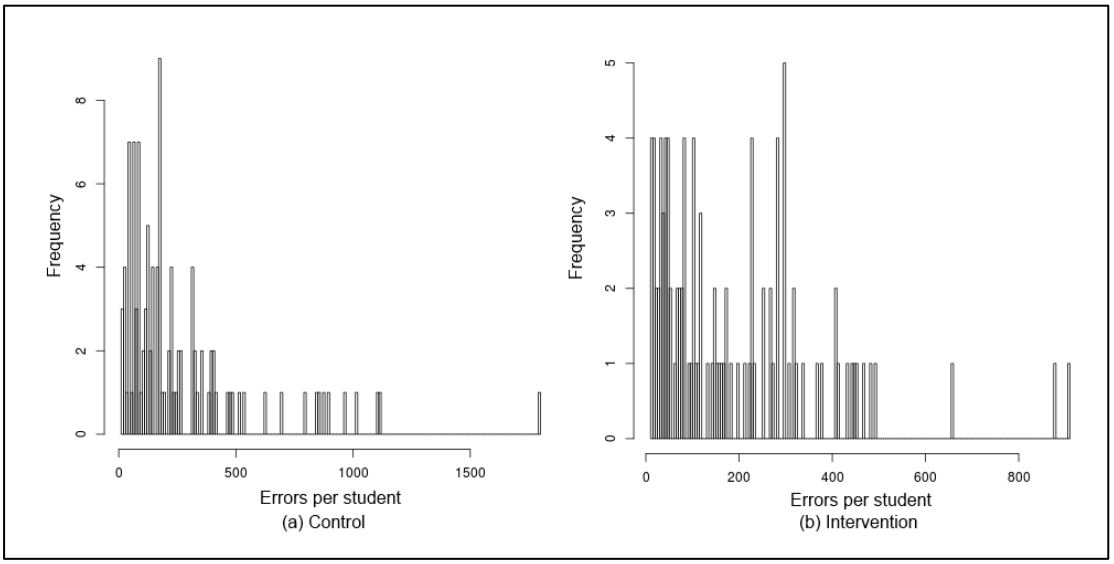

Figure 4.9 Histograms of errors per student (for all CEMs) for (a) control and (b) intervention. Note that the axes have different scales.

Figure 4.10 shows histograms for the same data, reduced to the top 15 CEMs. A Mann-Whitney *U* test (two-tail) showed that the number of errors was greater for the control group (*Mdn* = 152) than for the intervention group (*Mdn* = 125), *U* = 4,254, *p* = 0.028. As this finding is significant using a two-tail test, and the top 15 CEMs account for the majority of errors overall, this somewhat mitigates the one-tail/two-tail situation that occurred when investigating all CEMs. In other words we do not have to assume a reduction in errors for the intervention group, and resort to a potentially questionable

[14] The significance level α for all tests in this thesis is 0.05. *Mdn* is the median, *U* is the test statistic. If $p < \alpha$ the result is be considered significant.

choice of using a one-tail test, using the findings from the previous section as justification.

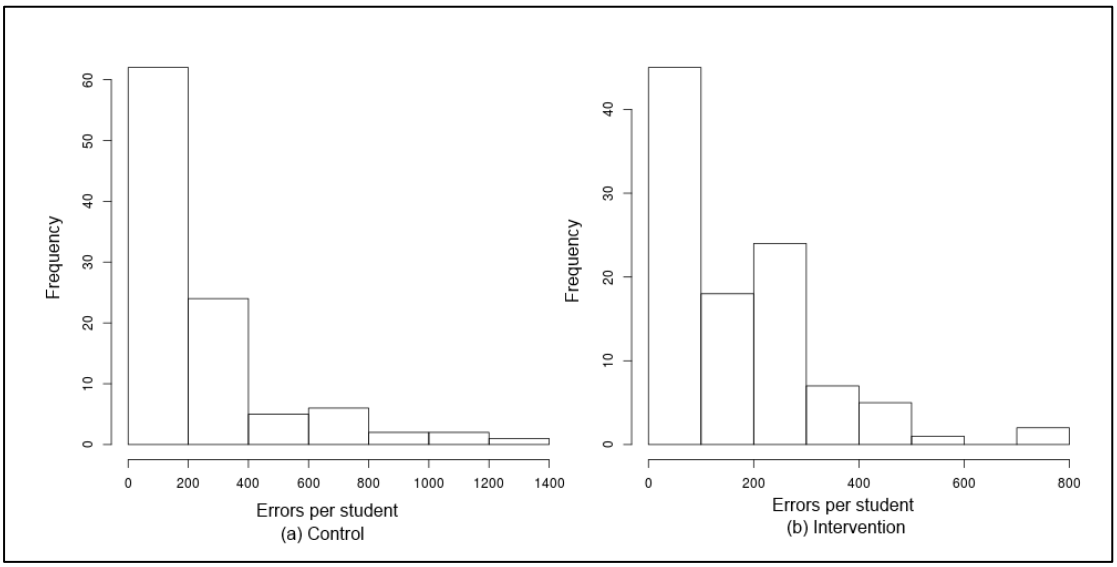

Figure 4.10 Histograms of errors per student (for the 15 most frequent CEMs) for (a) Control and (b) Intervention. Note that the axes have different scales.

Table 4.7 shows the nine of the top 15 CEMs that have a statistically significant difference in the number of errors per student between the control and intervention groups. Eight of these CEMs are enhanced by Decaf, accounting for 43.2% of all errors, while the one not enhanced by Decaf accounts for 1.9% of all errors. This is an extremely important finding – we have identified eight CEMs accounting for 43.2% of all errors, which when enhanced by Decaf reduce the number of student errors with statistical significance. Only one recent study investigated individual errors and reported no significant results for three (Denny, Luxton-Reilly, & Carpenter, 2014).

Table 4.7 Details of statistically significant top 15 CEMs (errors per student).[15]

CEM number	CEM description	Enhanced by Decaf?	Average, Median (Control)	Average, Median (Intervention)	Mann-Whitney *U* test (two-tail)
32	class, interface, or enum expected	Yes	9.9, 6.0	5.2, 3.0	$U = 3740$, $p = 0.001$
74	not a statement	Yes	21.0, 10.0	10.7, 6.0	$U = 3968$, $p = 0.003$
57	incompatible types OR incompatible types: *type* cannot be converted to *type*	Yes	9.5, 5.0	5.3, 2.5	$U = 4012$, $p = 0.005$
5	'.class' expected	No	5.1, 2.0	4.2, 0.0	$U = 4034$, $p = 0.006$
24	cannot find symbol	Yes	44.9, 35.0	33.0, 25.5	$U = 4148$, $p = 0.012$
1	'(' expected	Yes	5.0, 2.0	2.9, 1.0	$U = 4245$, $p = 0.023$
12	<identifier> expected	Yes	10.5, 4.0	4.5, 2.0	$U = 4330$, $p = 0.038$
51	illegal start of expression	Yes	17.9, 9.5	13.4, 7.0	$U = 4347$, $p = 0.042$
92	variable *variable name* is already defined in method *method name*	Yes	7.3, 4.0	5.0, 3.0	$U = 4351$, $p = 0.043$

CEM 5, which has a statistically significant difference, but is not enhanced by Decaf, is '*.class' expected*. There are several possible explanations for this. First, it could be a false positive. Second, there could be a genuine reason that intervention students committed this error with less frequency – perhaps a pedagogical difference between the semesters, although significant efforts were made to avoid any. Third, it is not known if helping students by enhancing some CEMs has a 'knock-on' effect of helping with other CEMs (which are not enhanced). Outlined below are the very beginnings of an investigation into just that. Figure 4.11 shows a Java program which defines an empty method (lines 6-7), which is called by the main method (line 4).

[15] The significance level α for all tests in this thesis is 0.05. *U* is the test statistic. If $p < \alpha$ the result is be considered significant.

```
1 class Test {
2    public static void main(String[] args) {
3         int a = 1, b = 2;
4         go(a, b);
5    }
6    public static void go(int x, int y) {
7    }
8 }
```

Figure 4.11 Java program to exemplify <*identifier*> *expected* and '*.class' expected* CEMs

Line numbers to the left are not part of the program. If line 4 is changed to `go(int a, b)`, CEM 5 *'.class' expected* is generated. This is because the type of the method parameter is already known. Remember, this is the one of nine CEMs *not* enhanced by Decaf in Table 4.6, but one for which the intervention group does have a lower number of errors for.

If `go(int x, int y)` on line 6 is changed to `go(int x, y)`, CEM 12 <*identifier*> *expected* is generated. This is because no type is given for `y`. This CEM *is* enhanced by Decaf and does have a significant reduction for the intervention group. Table 4.8 summarises this example.

Table 4.8 Comparison of '*.class' expected* and <*identifier*> *expected* CEMs.

Code	Comment	CEM	Enhanced by Decaf?
go(a, b)	Correct (line 4)	-	-
go(int a, b)	Error (line 4)	5 '.class' expected	No
go(int x, int y)	Correct (line 6)	-	-
go(int x, y)	Error (line 6)	12 <identifier> expected	Yes

Given the similarities between these two errors, it would not be entirely unreasonable to find that helping students with CEM 12 has a knock-on effect of helping them with CEM 5. Both errors can occur due to incorrectly stating (or not stating) the types of method parameters, in calling (line 4) or defining (line 6) a method. However these CEMs can arise in different situations and a proper investigation of this potential knock-on effect is beyond the scope of the present work.

4.2.1 Summary

This section sought to answer the questions: Do enhanced compiler error messages reduce the number of errors per student? If so, for what compiler error messages in particular?

It was shown that enhanced compiler error messages do reduce the number of errors per student, particularly for the following CEMs:

CEM number	CEM description
32	class, interface, or enum expected
74	not a statement
57	incompatible types OR incompatible types: *type* cannot be converted to *type*
5	'.class' expected
24	cannot find symbol
1	'(' expected
12	<identifier> expected
51	illegal start of expression
92	variable *variable name* is already defined in method *method name*

4.3 What effect do enhanced compiler error messages have on students, particularly students struggling with programming?

Insight to this question will be gained first by investigating repeated errors, which have been found to be an indicator of whether a student is struggling (Section 4.3.1). Then students will be examined as control and intervention groups, and their positions within their group (Section 4.3.2). Finally, individual student activity and behaviour will be examined through vignettes (Section 4.3.3).

4.3.1 Repeated Errors

To help answer the question posed above, we will first gain insight into the question:

Do enhanced compiler error messages reduce the number of repeated errors?

It is important to note that the number of errors a student commits is not a guaranteed measure of if that student is struggling (although a high number of errors is an indication that something may be wrong). It is possible to have a student compile frequently, committing many errors, in turn generating many CEMs, but resolving them quickly (and who is perhaps not struggling). On the other hand a student who is truly struggling may not even compile much and therefore commit few errors. For repeated errors it is a different situation - it is very possible to say that a student who commits an error repeatedly is struggling with CEMs. Jadud found that other than a higher frequency of errors, the number of repeated errors can be used to identify struggling students, with how often an error is repeated being one of the best indicators of how well (or poorly) a student was progressing. In fact his metric of error quotient (see Section 2.3) which was a measure of how well a programming student was fairing with syntax errors, is zero (good) if a student has no repeated errors, and high (bad) if there are a high number of repeated errors (2006).

A student is said to have committed a repeated error when two *consecutive in time* compilations result in the same CEM and originate from an error on the same line of code. Figure 4.12 shows the number of repeated error strings per student (by group) for the top 15 CEMs.[16] A repeated error string is an occurrence of at least one repeated error – it could be more than one repeated error, provided the repeated errors themselves are consecutive in time. Such a string ends when a different CEM is encountered or a different line of code causes the same CEM (each indicating that the original error was solved). A Mann-Whitney *U* test (two-tail) showed that the number of strings per student was greater for the control group (*Mdn* = 37) than for the intervention group (*Mdn* = 27), $U = 6437$, $p = 0.012$. Note that this data is not paired – each line in Figure 4.12 represents a succession of all students in each group, ordered in decreasing number of failed compilation strings. This shows that more control students made more repeated errors and were more likely to be struggling.

[16] See table 4.4 for full error descriptions.

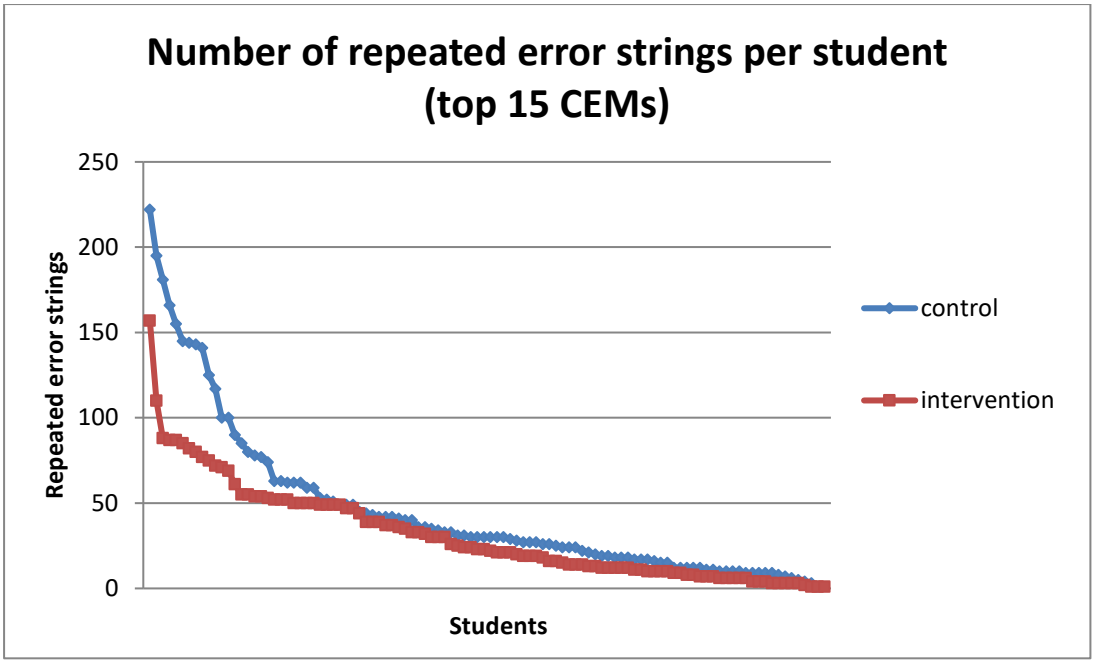

Figure 4.12 Number of repeated error strings per student (top 15 CEMs).

Figure 4.13 shows the number of repeated errors per CEM for the top 15 CEMs. A Wilcoxon signed-rank test (two-tail) showed that the number of errors was greater for the control group (*Mdn* = 742) than for the intervention group (*Mdn* = 416); *Z* = -2.90, *p* = 0.004.

In addition, a Shaprio-Wilk test showed that log transformed data was normal, and a paired two-sample *t*-test (two-tail) showed a significantly higher number of errors for the control group (*M* = 2.87, *SD* = 0.30) compared to the intervention group (*M* = 2.68, *SD* = 0.35); $t(14) = 4.73$, $p = 0.0003$.[17]

The only CEM with a higher number of repeated errors for the intervention group was 39 *else without if*. This was the only CEM in the top 15 with a higher number of (overall) errors for the intervention group (Figure 4.8), and one of the three top-15 CEMs (along with 5 and 53) that are not enhanced by Decaf.

[17] The significance level α for all tests in this thesis is 0.05. *M* is the mean, *SD* is the standard deviation, and *t* is the test statistic. If $p < \alpha$ the result is be considered significant.

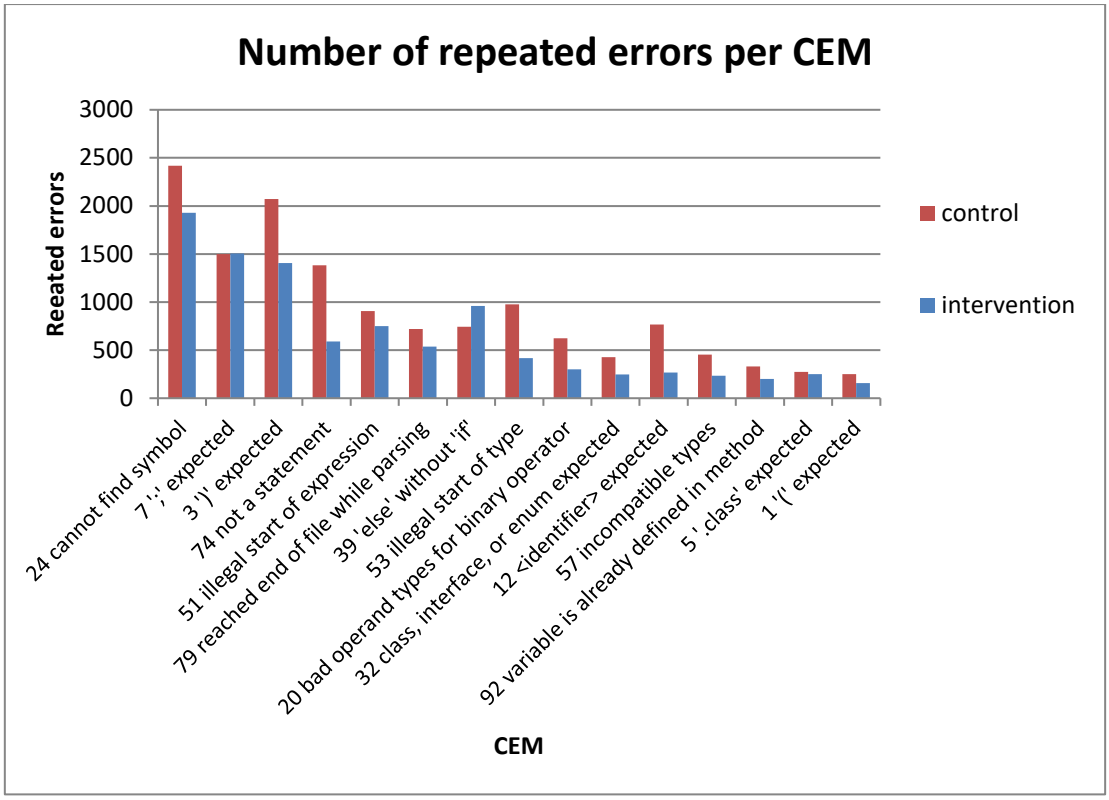

Figure 4.13 Number of repeated errors per CEM.

4.3.2 The Control and Intervention Groups and Their Students

We began this chapter discussing the control group, and comparing it to other studies. We then looked at data from an error perspective, followed by an error per student perspective. We then looked at how repeated errors affect students, as repeated errors have been shown to be a way to identify struggling students. We now take a step further by looking at the control and intervention groups, not as data on errors and students per se, but as groups of students.

Figure 4.14 shows the results of a principal component analysis (PCA) of a matrix of 15,688 values, each representing the number of times a particular student committed an error resulting in a particular CEM (74 CEMs × 212 students). PCA is a non-parametric method of reducing a complex data set to reveal hidden, simplified dynamics within it (Shlens, 2003). PCA takes as input a set of variables (which may be correlated) and converts them into a set of linearly uncorrelated principal components (PCs). The number of PCs is less than or equal to the number of original variables. PCA is useful for retaining data that accounts for a high degree of variance, and removing data which does not.

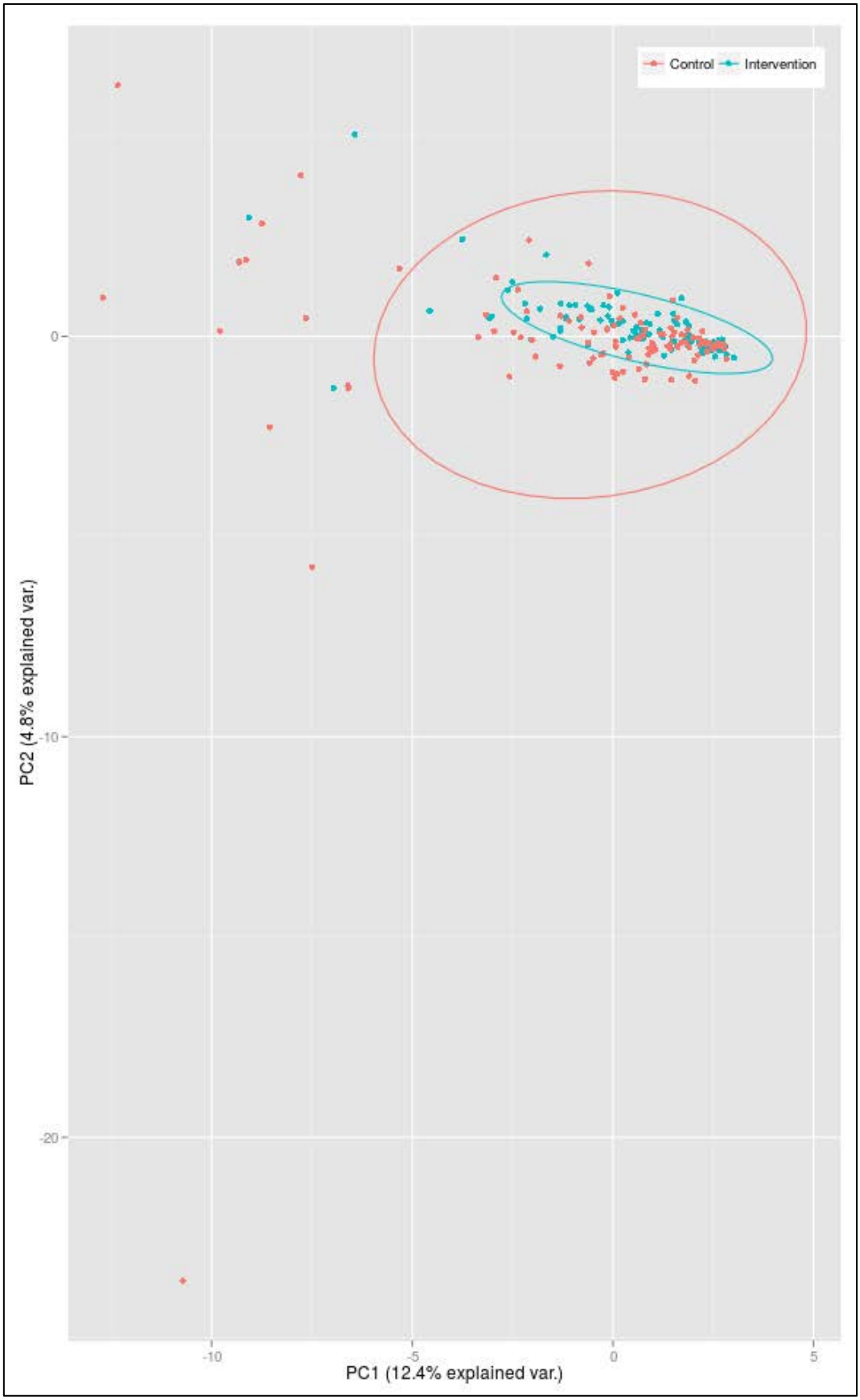

Figure 4.14 Principal component analysis of control and intervention groups, all CEMs.

The PCA was performed with the `ggbiplot`[18] function for the R statistical/graphical programming language. Each data point in Figure 4.14 represents a student. Groups are represented by different colours. The ellipses are 68% probability confidence ellipses. It can be seen that the intervention group exhibits less variance in principal components 1 and 2 (those with the greatest variance) as it has a smaller ellipse and along with having fewer, less distant outliers, it can be inferred that the intervention students are 'behaving' as a more cohesive group.

Outliers can influence the results of PCA, which is another reason that inactive students have been filtered from the data, and the further step of investigating the top 15 CEMs is taken. Again, it is believed that the data remaining is representative and useful, and that any outliers discovered are so for a reason which will help answer the questions this study investigates.

A PCA of the reduced (15 CEM) data is shown in figure 4.15. There are three immediate observations to be made:

1. The 'group profiles' remain very similar.
2. Individual students do not vary much (labels have been removed from the figures for clarity, but they are identifiable with the labels turned on).
3. The variance of the PCs increase substantially (PC1 from 12.4 to 41.1% and PC2 from 4.8 to 13.8%). Thus for the reduced (15 CEM) data, PCs 1 and 2 account for 53.5% of the variance.

It is important to note when comparing Figures 4.14 and 4.15 that the direction (positive/negative) of the PCs and the resulting correlation with variables (CEMs) is arbitrary, so for instance the fact that the outlying student beyond (-10, -20) in Figure 4.14, is located beyond $y = 15$ in Figure 4.15, does not represent anything of interest in and of itself, as all students have been shifted accordingly between the figures. It is the relative position of students within each figure, not between the figures, that is of interest.

[18] github.com/vqv/ggbiplot

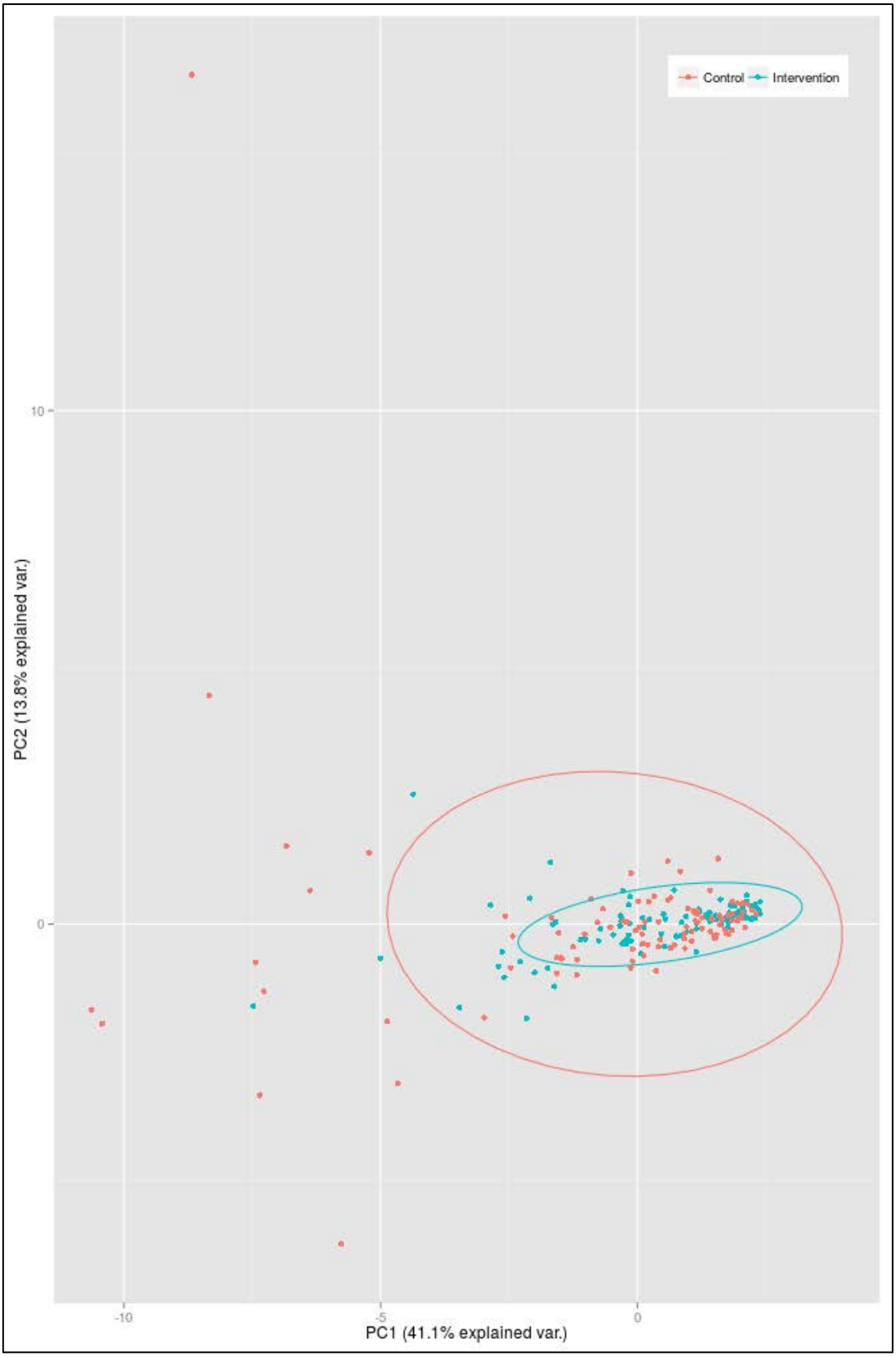

Figure 4.15 Principal component analysis of control and intervention groups, top-15 CEMs.

Table 4.9 shows the Pearson Correlation Coefficient of each CEM with each principal component.

Table 4.9 Correlation with the first five principal components with the top 15 CEMs.

CEM	PC1	PC2	PC3	PC4	PC5
1	-0.60	-0.13	0.01	0.00	-0.06
3	-0.68	0.55	-0.22	0.20	0.00
5	-0.54	-0.16	-0.12	0.47	-0.58
7	-0.79	0.00	-0.15	-0.15	-0.04
12	-0.37	0.06	-0.25	0.17	0.21
24	-0.78	-0.20	-0.10	-0.09	0.18
51	-0.88	0.00	0.15	-0.01	-0.15
74	-0.58	0.01	0.23	-0.63	-0.33
79	-0.68	0.00	0.41	-0.12	-0.03
92	-0.59	-0.30	-0.30	-0.17	0.32
32	-0.66	-0.20	0.42	0.24	0.24
20	-0.56	-0.30	-0.51	0.09	-0.24
39	-0.60	-0.10	0.46	0.37	0.24
53	-0.57	0.05	-0.12	-0.12	0.03
57	-0.57	-0.40	-0.52	-0.11	0.23

This information reflects correlations between the PCs and the original variables (CEMs). Interpretation of the PCs is based on finding which variables are most strongly correlated with which component, and is a subjective decision. For instance, we can see that all CEMs are negatively correlated to PC1 and further, all PCC values other than CEM 12 are below -0.50. As CEMs 7, 24 and 51 are the only three below -0.75 it can be taken that PC1 is correlated with these CEMs very strongly (the fact that they are negative is arbitrary), suggesting that these three CEMs vary together.

This 'PCA perspective' of each group, along with the individual student vignettes in the next section provide an illustrative bridge serving to enhance the triangulation between the quantitative data presented in Sections 4.1 and 4.2 and the qualitative student and educator data in (or referenced from) Section 4.4. We will have a slightly more detailed look at this PCA data in the next section. A much more detailed analysis is possible but is beyond the scope of this thesis.

4.3.3 Vignettes

To gain a view on the activity and experience of individual students this section presents vignettes of four students, chosen for their relative positions on the PCA plot shown in

Figure 4.16, and for qualities discovered in their profiles, summarised in Table 4.10 and discussed individually below.

Table 4.10 Profiles of vignette students.

	Student compiler ID (Group)							
	1196* (Control)		144* (Intervention)		1142 (Control)		107 (Intervention)	
CEM	*Errs*	RErrs	*Errs*	RErrs	*Errs*	RErrs	*Errs*	RErrs
24	*62*	26	*60*	39	*69*	44	*76*	46
7	*69*	28	*71*	44	*18*	7	*37*	20
3	*488*	449	*296*	278	*82*	56	*17*	10
74	*11*	7	*32*	23	*25*	16	*13*	4
51	*63*	26	*47*	33	*13*	7	*2*	0
79	*7*	3	*23*	17	*9*	5	*28*	13
39	*12*	4	*78*	68	*4*	3	*13*	10
53	*320*	285	*29*	23	*6*	3	*22*	18
20	*4*	0	*13*	3	*5*	2	*1*	0
32	*3*	1	*11*	6	*21*	14	*4*	1
12	*321*	285	*22*	15	*11*	6	*16*	12
57	*7*	2	*7*	2	*2*	1	*8*	5
92	*10*	6	*1*	0	*7*	4	*1*	0
5	*9*	3	*1*	0	*12*	8	*4*	1
1	*7*	1	*14*	9	*3*	0	*2*	1
Total	*1393*	1126	*705*	684	*287*	176	*244*	141
RErrs/Errs	81%		79%		61%		58%	
Total (for all CEMs)	1797	1147	876	560	324	195	307	178

* = Outliers on PCA plot
Errs = Errors | RErrs = Repeated errors
Figures in red are discussed in this section specifically as being relatively high. Figures in green for being low.

Figure 4.16 also includes variable factor arrows representing the correlation of CEMs with the first two principal components. Figure 4.17 shows this area in greater detail. The angle between any two arrows represents the correlation between those CEMs (90% is linearly uncorrelated). The purple circle represents the theoretical maximum extent of the arrows. Over 90% of students lie within this circle, and this could be used for fine grained comparisons and analysis of students close to the centres of their groups. This is beyond the scope of this thesis, but is an intended area for future work. Nonetheless, can provide an even deeper insight into the behaviour of the groups and their students.

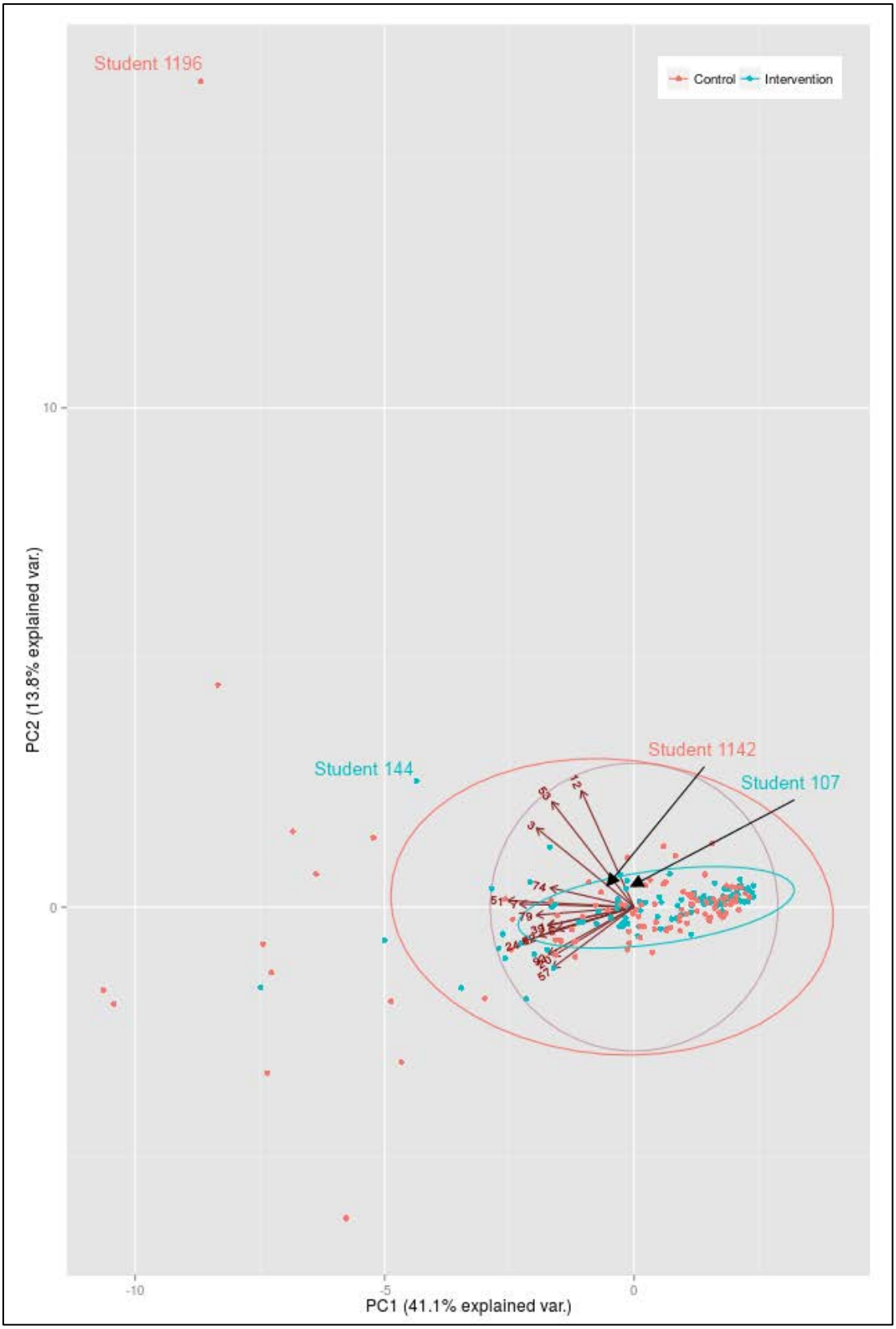

Figure 4.16 Principal component analysis for the top 15 CEMs showing positions of the four vignette students discussed in this section.

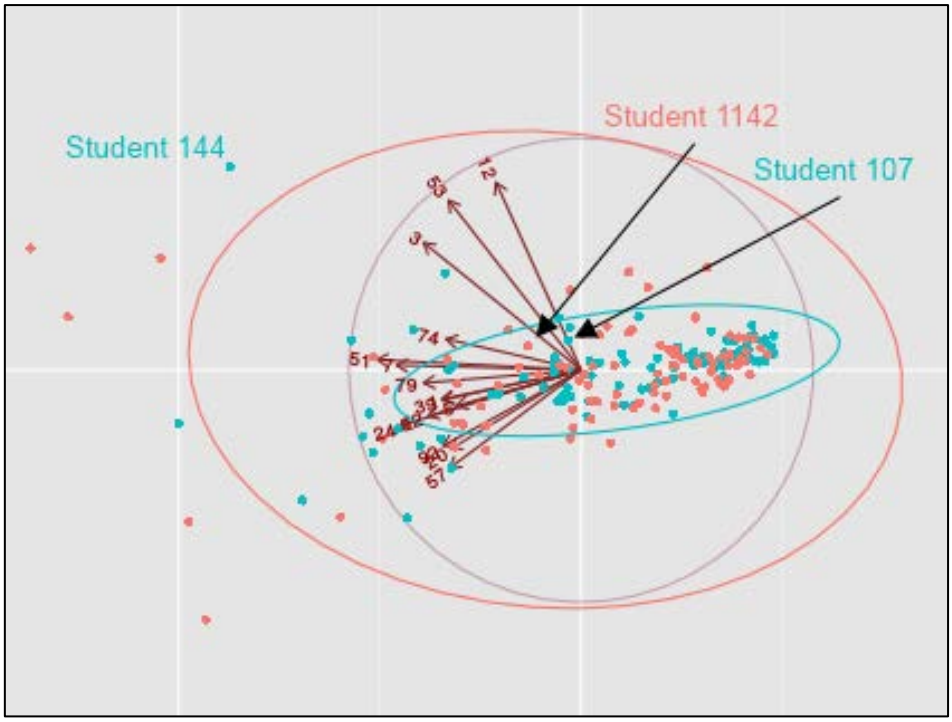

Figure 4.17 Detail from Figure 4.16.

We will begin by looking at all four students together, and then look at their activity profiles and behaviours separately. Figure 4.18 shows the errors per CEM for our four students. CEMs are ordered in decreasing overall frequency (across both groups as presented in Figure 4.8). Student 1196 (control group, PCA outlier) has very high relative frequencies of CEMs 3, 53, and 12. Student 144 (intervention group, PCA outlier) also has a high number of errors for CEM 3.

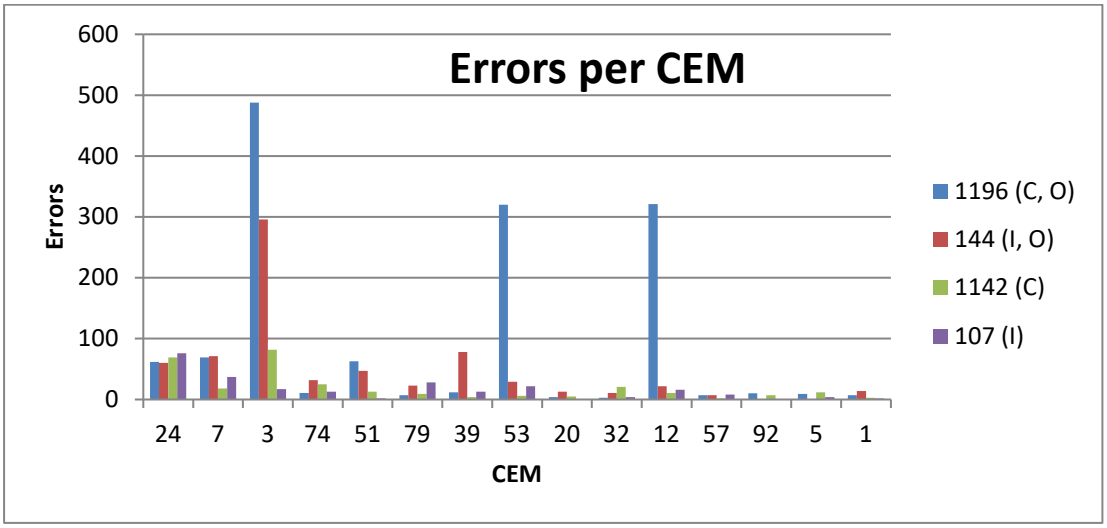

Figure 4.18 Errors per CEM for the four vignette students. C: control group, I: intervention group, O: PCA outlier.

Figure 4.19 shows the same plot, but with a log scale on the *y*-axis to show details for lower frequencies. It is notable that these students (with the exception of some CEMs for outlier students) somewhat follow the trend seen in Figure 4.8, particularly student 107. In addition, student 144 has a relatively high number of errors for CEM 39.

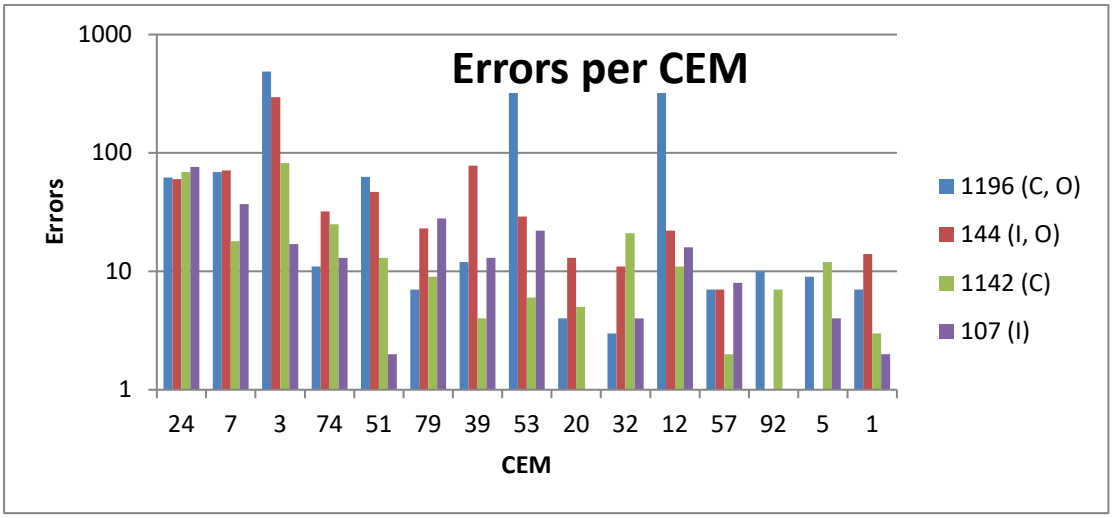

Figure 4.19 Errors per CEM (log scale on *y*-axis) for the four vignette students. C: control group, I: intervention group, O: PCA outlier. Note that students with 0 or 1 error for a given CEM do not feature as $\log_{10}(0)$ is undefined and $\log_{10}(1) = 0$.

Figure 4.20 shows the repeated errors for our four students. Again, student 1196 has an extremely high number for CEMs 3, 53 and 12, and student 144 is also high for CEM 3. Perhaps the most notable feature however is the ratio of repeated errors to overall errors. Table 4.9 reveals that the two outlier students have 79 and 81% repeated errors compared to 58 and 61% for the two students very near the centre of their groups (0,0) on the PCA plot (Figure 4.16). This suggests that one of the criteria of outlying students is a high ratio of repeated errors to overall errors.

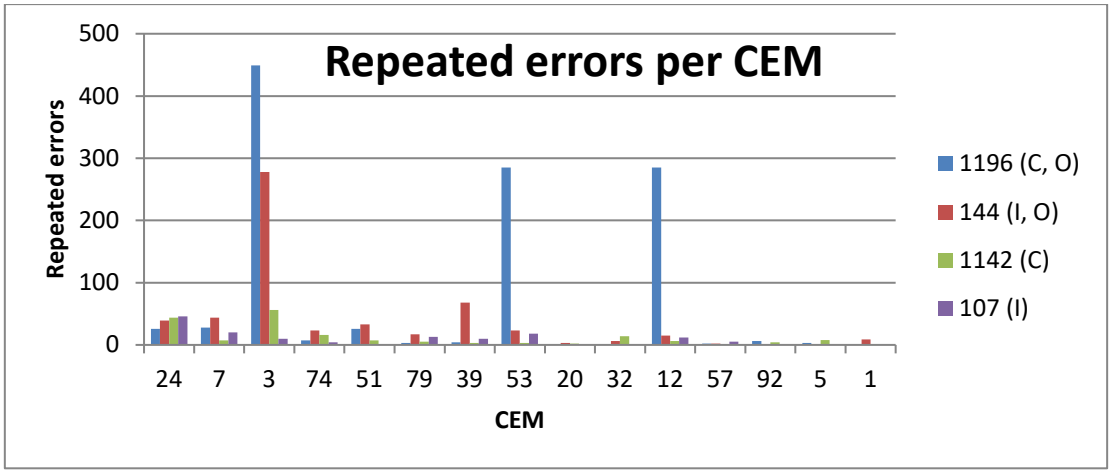

Figure 4.20 Repeated errors per CEM for the four vignette students. C: control group, I: intervention group, O: PCA outlier.

Figure 4.21 shows the same plot, but with a log scale on the *y*-axis to show details for lower frequency errors. This reveals that student 144 has a relatively high number of repeated errors for CEM 39, as is the total number of CEMs for this student.

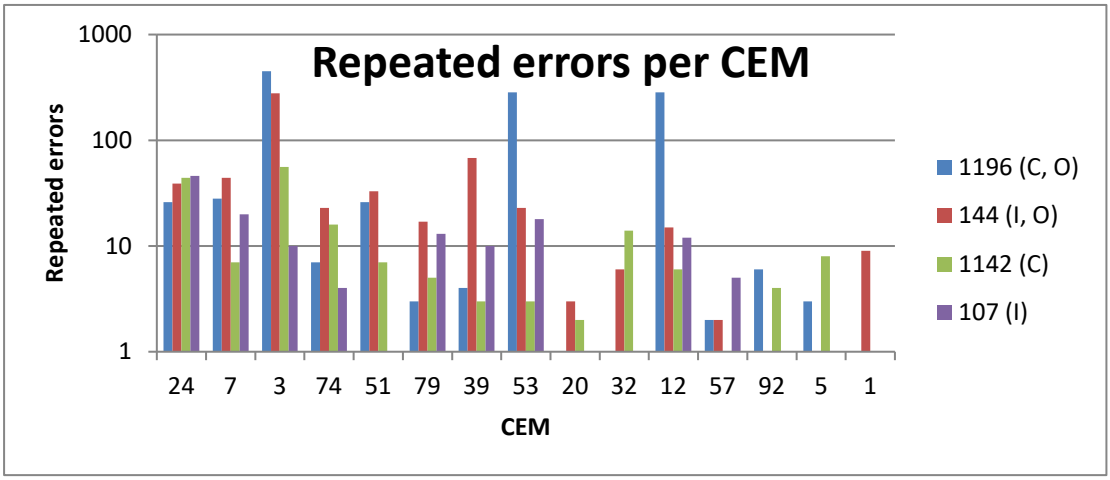

Figure 4.21 Repeated errors per CEM (log scale on *y*-axis) for the four vignette students. C: control group, I: intervention group, O: PCA outlier. Note that students with 0 or 1 error for a given CEM do not feature as $\log_{10}(0)$ is undefined and $\log_{10}(1) = 0$.

.Student 1196 (Control, Outlier)

Figure 4.22 shows the error profile of student 1196 (control group). It was hard to find a single session that did not have a significant number of repeats. One session is defined as all errors working on one file (program). This student had many long strings of repeated errors for CEMs 3, 9, 12, and 53. Interestingly, this student's position on the PCA (Figure 4.16) corresponds well with the arrows for CEMs 3, 12 and 53, despite the fact that this student lies outside the circle. It is also noteworthy that this student has relatively few errors for CEMs 24 and 7, the two most frequent errors across all students. As Figure 4.21 is quite busy, Figure 4.23 shows one of this student's sessions, with a high number of repeated errors for CEMs 3, 9, 12 and 53. This student had a repeated/total error ratio of 81% for the top 15 CEMs, the highest of all four students studied here.

Note: In the remaining figures for this section, errors (on the x-axis) are presented in the order they occurred, but they are evenly spaced. In other words the x-axis does not represent time between successive errors, only their order.

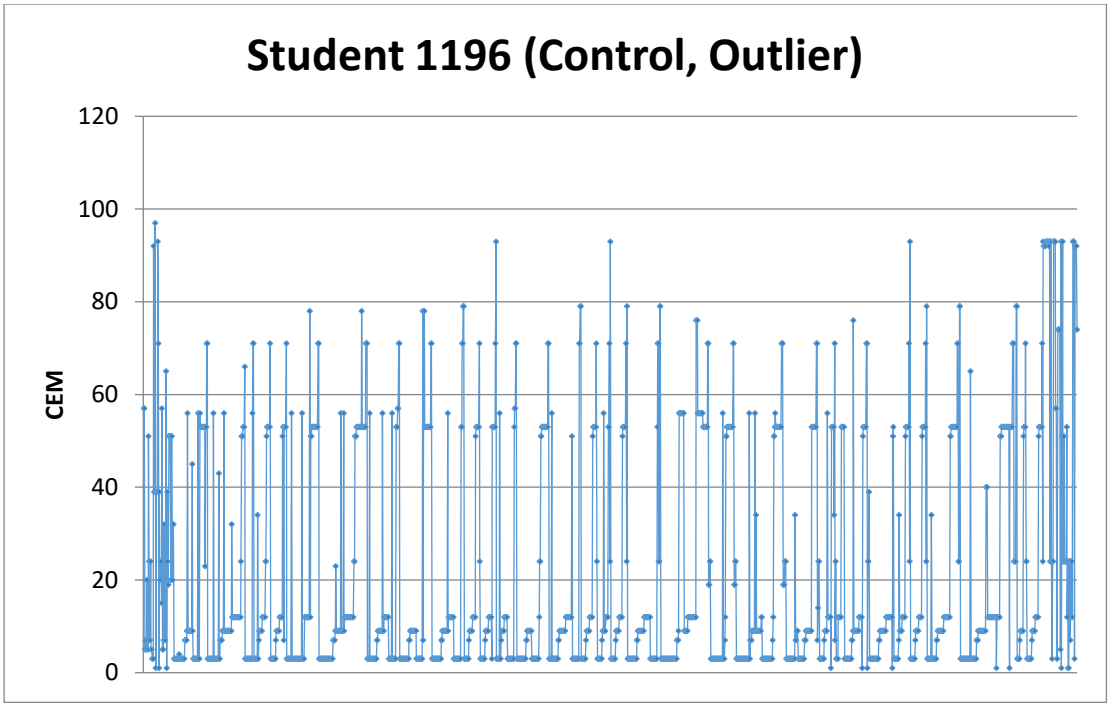

Figure 4.22 Error profile for student 1196.

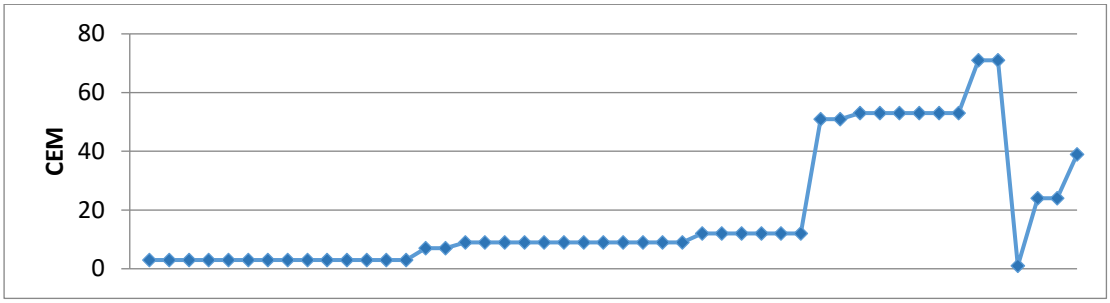

Figure 4.23 Error profile for one session (time working on one file), student 1196.

Student 114 (Intervention, Outlier)

Figure 4.24 shows the activity profile of student 114 (intervention group). This is notable as this student really struggled with errors 3 and 85 early on. However as time progressed the amount of repeated errors reduced significantly, as did the number of CEMs encountered. This student had disproportionately high frequencies for CEM 39 and low frequencies for 7 and 24. This student had a repeated/total ratio of 79% for the top 15 CEMs, very nearly that of Student 1196, the other outlier, at 81%.

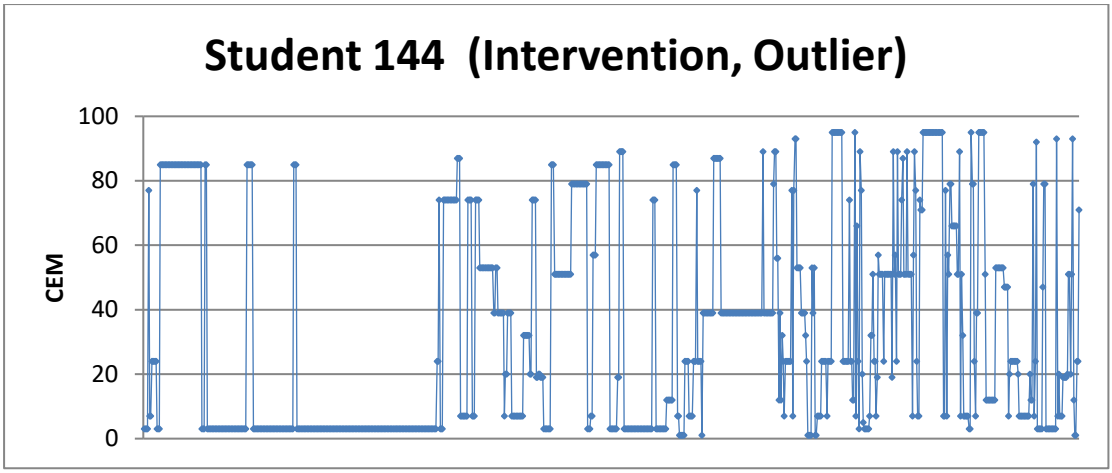

Figure 4.24 Error profile for student 144.

Student 1142 (Control)

Figure 4.25 shows the activity profile for student 1142 (control group). This student had relatively normal error frequencies, and a repeated/total error ratio of 61%, approximately 20% lower than the outlier students.

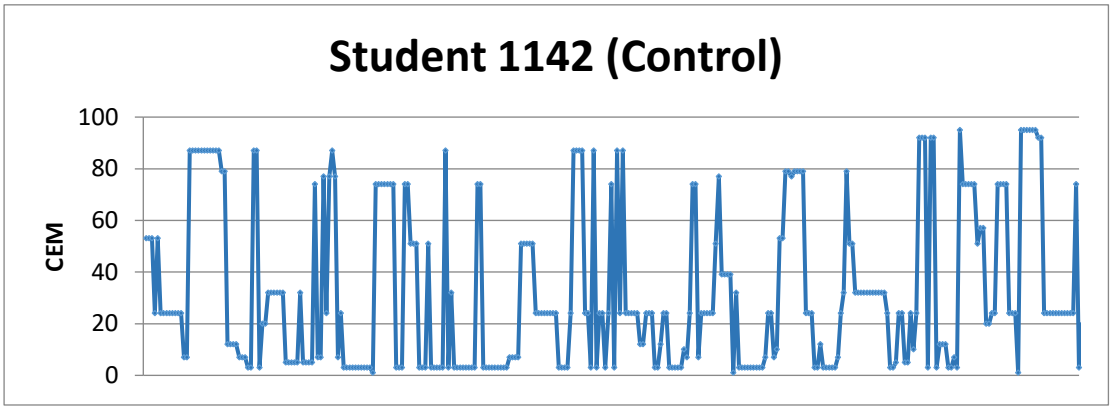

Figure 4.25 Error profile for student 1142.

Student 107 (Intervention)

Figure 4.26 shows the activity profile for student 107 (intervention group). This student had relatively normal error frequencies, and a repeated/total error ratio of 58%, approximately 20% lower than the outlier students and very similar to student 1142, the other non-outlier.

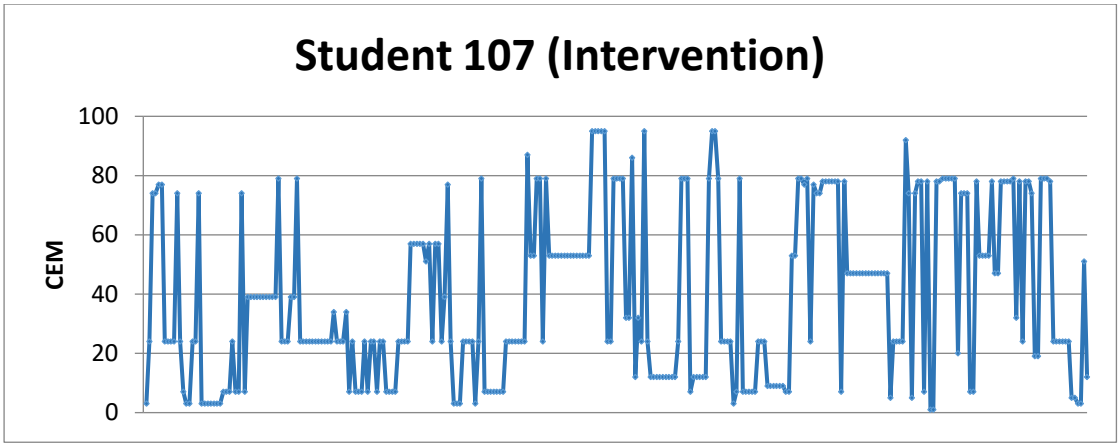

Figure 4.26 Error profile for student 107.

Summary

This section and Section 4.3.2 presented detailed profiles of both groups and of four individual students. One of the most important observations is that the intervention group has behaviour which is more cohesive than the control group – that is in terms of their error profiles intervention students are more similar to each other than are students in the control group. The approximate ratio of ellipse areas (control:intervention) is 6.5:1 in Figure 4.14 (representing all CEMs), and 4.5:1 for Figure 4.15 (representing the top 15 CEMs). In addition, considering that the centre of both groups is (0,0), if a circle with radius 2.5 and centre (0,0) is superimposed on Figure 4.14, 14 control but only 3 intervention students lie outside. For Figure 4.15 these numbers are 13 and 3 respectively, with 2 of these 3 intervention students extremely close to the circle. The two 'outlying' students investigated in this section are outside of this circle on both figures.

Investigating four individual students in this section gives an insight to factors that may contribute to a student falling far from the centres of their groups:

1. A high number of errors relative to other students.
2. A high number of repeated errors to overall errors.
3. An error profile that does not follow the 'normal' trend for all students, shown in Figure 4.8.

Combined, these factors serve as evidence that Decaf can help reduce the number of struggling students, and 'unite' groups more tightly.

4.3.4 Summary

This subsection sought to answer the question: *What effect do enhanced compiler error messages have on students, particularly students struggling with programming?*

Enhanced compiler error messages were found to have a positive effect on repeated errors, a key indicator of if a student is struggling. A detailed look at both groups showed that ECEMs resulted in the intervention group behaving more coherently with fewer signs of struggling students. Finally individual vignettes demonstrated experiences from both groups. Although in most cases it would be impossible to determine simple from a student's activity profile if that student was experiencing enhanced error messages or not, vignettes were found which exemplified the difficulties that students can have with compiler error messages, and how enhanced error messages can help them overcome these difficulties.

4.4 What are student and educator views on enhancing compiler error messages?

To reveal insight to this question, student surveys presented in Section 4.4.1, and lecturer interviews presented in Section 3.3.2.2 were conducted. This completes the journey from 'hard data' on errors, to errors per student, through group behaviour and in the last section individual student data, activity and profiles. We now come into contact with students first-hand, learning about their experiences, opinions, and hearing their own words.

4.4.1 Students

To get an idea of how learners viewed and experienced compiler error enhancement with Decaf a short optional and anonymous survey was employed. Details of the design are included in Section 3.3. The survey was comprised of four mandatory Likert questions, each with an optional open-ended field asking "Please explain (optional)". There were also four (stand-alone) open-ended questions to illicit deeper feedback on Decaf and the student experience of using it.

The response rate was approximately 32% for the intervention group and 20% for the control group. It is interesting to note that for the intervention group, an average of 28% of the optional comments were completed compared to 7% for the control group. These numbers are one indication that the intervention group was much more vocal about their

experience. An independent-samples *t*-test (two-tail) was conducted for each Likert question.

4.4.1.1 Likert Questions

Question 1

Figure 4.27 shows the first question:

How frustrating do you find compiler errors?

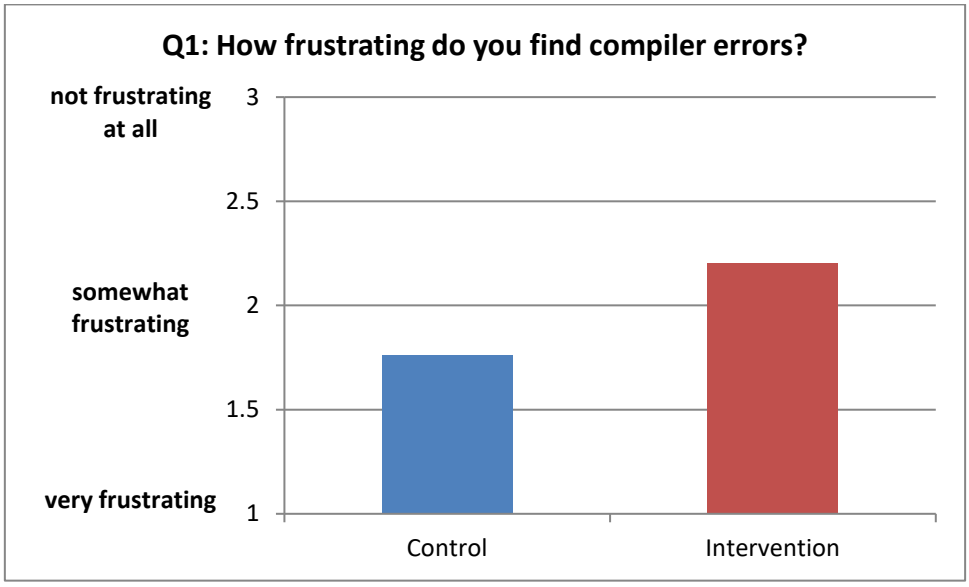

Figure 4.27 Q1: How frustrating do you find compiler errors?

There was a significant difference in the responses: control (M = 1.76, SD = 0.61), intervention (M = 2.16, SD = 0.72); $t(51) = 2.11$, $p = 0.040$.[19] This indication is encouraging, particularly as the intervention students were aware that they were being presented with both the javac CEMs as well as the Decaf ECEMs. It was possible that students would find being presented with two error messages instead of one confusing, or frustrating. Below are selected open-ended comments from this question:

- Control
  - "If you know how to read the compiler error message, it is easy to find the error"
  - "The way the errors are presented to the user are not very easy to read, but I assume it might be a java standard"

[19] The significance level α for all tests in this thesis is 0.05. M is the mean, SD is the standard deviation, and t is the test statistic. If $p < \alpha$ the result is be considered significant.

- Intervention
    - "Not frustrating at all because I'm still learning how to write the codes and Decaf helps me a lot to not make same mistakes next time"
    - "Sometimes I don't know what the errors mean"
    - "Not frustrating because it shows you where the errors are"

Question 2

Figure 4.28 shows the second question:

How much of a barrier to progress do you feel compiler errors are?

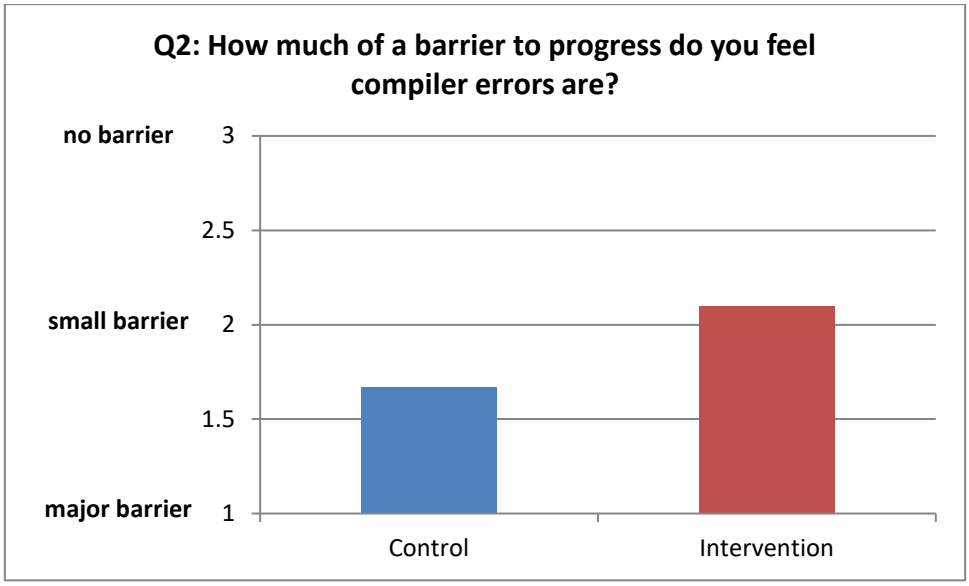

Figure 4.28 Q2: How much of a barrier to progress do you feel compiler errors are?

There was a significant difference in the responses for control (M = 1.67, SD = 0.64) and intervention (M = 2.09, SD = 0.72); $t(51) = 2.21$, $p = 0.032$. Below are selected open-ended comments from this question:

- Control
    - "Sometimes it might be a little error like a bracket but then the compiler comes up with many errors"
    - "gives us extra stress and it makes programming harder than it actually is"
- Intervention
    - "I wouldn't say a barrier to progress, I think this helps me a lot to write myself the codes and encourages me to study more"

- "Compiler errors are a great tool to finding errors and problems within your code"
- "Small barrier because it directs you in what to do to solve a problem"

Question 3

Figure 4.29 shows the third question:

Would you recommend Decaf to someone who wants to learn Java but has never programmed before?

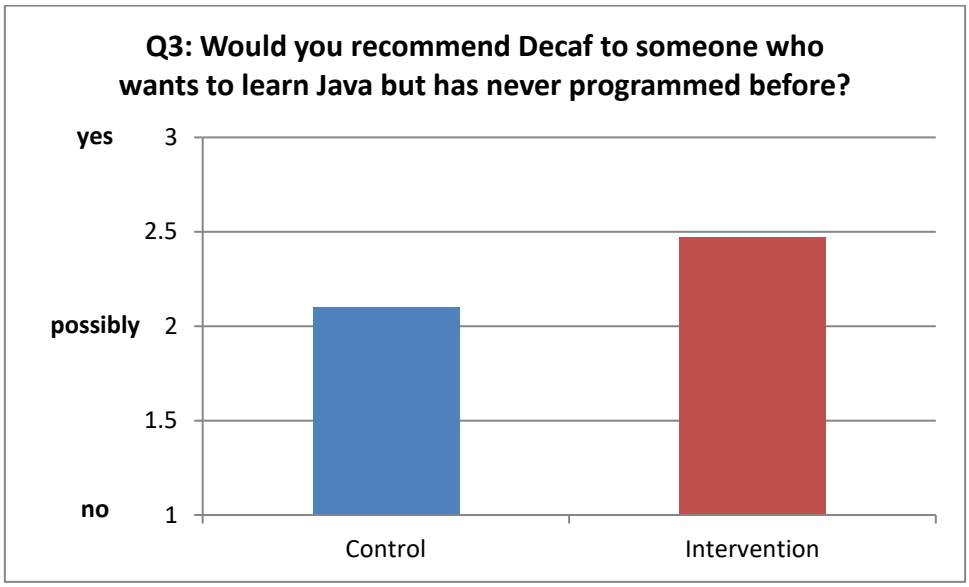

Figure 4.29 Q3: Would you recommend Decaf to someone who wants to learn Java but has never programmed before?

There was no significant difference in the responses for control (M = 2.15, SD = 0.57) and intervention (M = 2.49, SD = 0.61); $t(50) = 1.86$, $p = 0.070$. In the open-ended section, 14 intervention students wrote additional comments stating that they would recommend Decaf. One intervention student said they would not, and recommended notepad++ (another editing tool). No control students left open-ended comments. Below are selected open-ended comments from this question:

- Intervention
  - 'Yes. Because I find that decaf is very helpful for first time programmers like me"
  - "Because is much easier to understand error in English than straight in Java language and it helps a lot in my opinion..."

- "It is a very good help for gaining the understanding of compiler errors. Easy to use and to grasp the fundamentals of Java without having to worry about how to get started in more complex IDE's."
- "Yes I will recommend Decaf because it explains a lot"
- "Yes, because it gives you details about the errors also it has a friendly interface"

Question 4

Figure 4.30 shows the fourth question:

On the following scale, how much easier do you think Decaf makes learning to program?

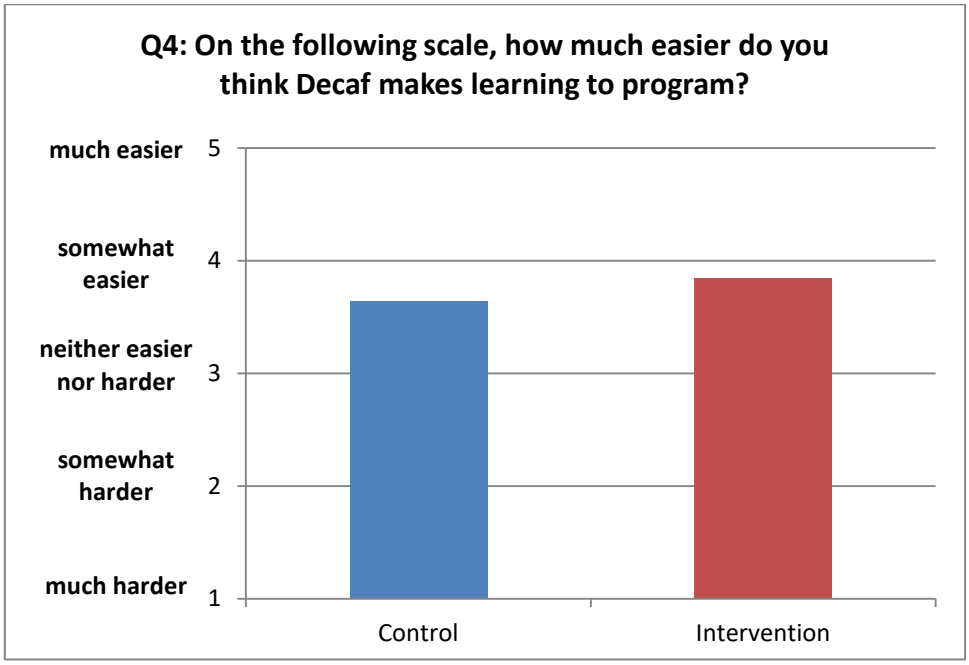

Figure 4.30 Q4: On the following scale, how much easier do you think Decaf makes learning to program?

There was no significant difference in the responses for control (M = 3.64, SD = 0.88) and intervention (M = 3.84, SD = 1.00); $t(41) = 0.625$, $p = 0.540$. The control group left no open-ended comments. Below are selected open-ended comments from the intervention group:

- Intervention
  - "I find it much easier to identify the error in English"
  - "It shows what you are doing wrong"
  - "Decaf is great for beginners"

- "It is an amazing way to learn"
- "Decaf helps me avoid small mistakes"

4.4.1.2 Open-ended Questions

Question 5: ***What did you like about Decaf?***

- Control
  - "Clean interface"
  - "Other softwares like netbeans, eclipse are heavy and occupies the most of the screen with so many things but decaf is like a fresh page. It has a nice and cute look."
- Intervention
  - "Decaf is a user-friendly software, and by using it we get less errors"
  - "It makes programming a little easier by showing you the errors"

Question 6: ***What did you not like about Decaf?***

Both groups responded with details about bugs, the most frequent one being a problem with infinite loops causing Decaf to hang. I have rectified this problem. Another comment was about non-English keyboards, which has also been rectified.

Two intervention students wrote positive comments, restating what they *did* like, and the wording of the comments indicate that they *knew* they were answering a question asking on what they did not like.

Question 7: ***How can Decaf be made better?***

Most comments were similar to the responses for Question 6. Two intervention students included comments that are believed to be about the decaf ECEMs:

- "Maybe adding more accurate solution suggestion/detection with compiler errors"
- "Being more specific"

The fact that at least two students want *better* ECEMs can be taken as evidence that they are viewed positively.

Two control students included comments *wanting more* from the javac CEMs:

- "Find bracket errors"
- "To be exactly precise what are the errors and what to be done"

Question 8: ***How many compiler errors do you think you generate yourself, per week?***

As this was an open-ended question, the responses are obviously open to any value, and cannot be treated with any confidence statistically – many comments were of the form "plenty", "very few", etc. However it is interesting that a total of 15 intervention and 7 control students did include specific values or ranges such as "more than 200". I casually extracted this information, and the control group averaged approximately 500, while the Decaf group averaged approximately 50. This is definitely to be taken with a pinch of salt, but it is interesting that this indicates that the intervention group thinks that they made fewer errors, particularly as this is one of the main goals of Decaf. In addition, looking at these comments overall it seems that the intervention students were much more specific with their estimates. The most any intervention student said was "about 150". The control students on the other hand reported:

- "hahahaha! a lot"
- "Lots!"
- "Many…"
- "Let's just say thank god numbers are infinite"

Two intervention students had comments relating to how they learn from CEMs, and how Decaf helps:

- "Many for sure, but it is good to make mistake because we learn a lot from them"
- "Toooooo many, but having errors is a good way to learn"

4.4.2 Educator

Findings

Below I present the findings of both interviews, providing insight to the experience of both groups, at times comparing and contrasting them. The findings are structured on the points above. Comments from the interviews themselves are referred to by number (C*x* for control interview comments and I*x* for intervention interview comments). The comments themselves are listed in Appendix G.

a. Lecturer experience
   i. Methodology

The lecturer carried out the module with both groups in a common manner, and similar to how he would outside of this study. He noted that the control semester was ‘pretty normal’ (C6). When prompted on assignments he noted that both groups used Decaf for assignments and labs almost exclusively. He commented on the control group using Decaf similar to any other editor, and any issues that arose being ‘normal’ (C1), noting that some students took longer than others to adjust to the software, but no more than he has seen with other IDEs (C2). He said the switch from Decaf to EJE was smooth for the control group (C7), and that some students continued to use EJE (C8). In the intervention group, he noticed that several students kept using Decaf after the study period (I9) and that some students used both Decaf and EJE together. One student used both for the same programs, copying and pasting back and forth, seemingly trying to get the best of both editors (I10).

ii. How did it differ from the ‘norm’

The lecturer noted that the transition from Decaf to EJE was time consuming for the intervention group (I8) and indicated that he was getting more than the normal amount of questions for such a process. He said that the intervention students were more interactive than a normal group, asking “a lot” of questions, and the transition from lecture to lab was longer and more interactive than he was used to (I9, I12). In the control interview he stated that he knew he should only be discussing the control group, but mentioned that he felt that the control period would have been smoother if the control students had the full version of Decaf (which enhances CEMs) (C11).

b. Student experience

i. How did it affect student behaviour (compilation)?

The lecturer commented that intervention students seemed to enjoy using Decaf and he noted that on occasion students were asking each other questions about and comparing their errors (I1). He also noted that students seemed to be engaging in more group work and interaction, particularly in lab sessions (I7). When prompted on comments from students, he had a lot to say noting that the intervention students made comments on errors that Decaf does not enhance, and one student asked if he could have the source code for Decaf so he could see how it works and if he could improve it. This level of interest was noted as quite welcome and surprising. He also received questions specifically on syntax errors from the intervention group and interestingly on the *causes* of errors (I3) this was mentioned again later (I13). This is noteworthy as it is more

common for students to only be concerned with how to resolve errors. This matched with another comment (I4) where he noticed that students named Decaf in their questions. Normally students refer to their editors as 'the compiler' or some other colloquial term. It was discussed that this could be due to him calling Decaf by name often in class. He ended by saying that some of the intervention students continued to use Decaf after the study period (I6). There was little to say about the control group, other than a lot of concern over syntax, which is common in a beginner programming course (C4). A similar comment was reiterated later (C5). When prompted about comments from students he had no further comments.

ii. How did students interact with Decaf? Positively/Negatively?

The lecturer commented that the intervention group was quite positive about Decaf and that they were impressed (the word flattered was mentioned) that the software was written by CCT staff, with their interest in mind (I5). The use of the word flattered did alert me that there was potential bias. He also mentioned that he spent more time with students in groups than normal, often himself and a handful of students all discussing code on one student's screen (I2). He noted that some of the control group students chose to use other IDEs, perhaps three or four students in total, who worked together often. He wondered at one point if this would spread throughout the group, but it did not (C3).

c. Improvements

Responses on improvements for Decaf were very consistent for both groups.

i. Software

The lecturer noticed that decaf was giving some students problems with non-English keyboard layouts (C9, 12) but that this was not uncommon for other non-professional editors he has used. There was also a problem with installing Decaf on OSX (C13). He noted an interest amongst intervention students when an infinite loop caused Decaf to hang. The students became quite interested in this and in one lab session were trying to replicate the problem (I11, 14).

ii. Use

The lecturer said that in general Decaf worked well and that having a minimal amount of options (less than EJE and much, much less than Eclipse) was beneficial to novice

students. He is used to getting many questions in other classes when using Eclipse, often related to features that students have access to but do not need (C10).

4.4.3 Summary

This subsection sought to answer the question: *What are student and educator views on enhancing compiler error messages?*

Questionnaires and interviews revealed that student difficulties with compiler error messages are significant and palpable in the classroom. The students that experienced enhanced error messages reported a positive experience with many believing that enhanced error messages helped raise the barrier posed by compiler error messages without enhancement.

4.5 Threats to Validity and Limitations

Before concluding this chapter a discussion on threats to validity is warranted. The first and most obvious is that the control and intervention groups were separated by a year. Attempts were made to make all environmental and pedagogical factors as similar as possible. Students learned the same topics in as similar a way as possible, experiencing the same lecturer, material, labs and environment. Nonetheless some factors could not be controlled such as scheduling differences, room availability and external pressures on students from other modules.

A more technical threat to validity is the fact that a new anonymous compiler ID is issued when Decaf is reinstalled, perhaps by the same student on the same computer, or by one student on multiple computers. This creates an issue in not having a perfect one-to-one mapping of compiler IDs to students. It is believed that this did not impact the results to a high degree for two reasons. First, the number of compiler IDs was not much above the average attendance and the average number of students submitting lab exercises. Second, filtering data to remove inactive compiler IDs brought the number of compiler IDs closer to the expected numbers, and filtering to remove infrequent CEMs demonstrated that the control group had a similar error profile to other studies.

This threat will be mitigated in future versions by persistently storing the compiler ID in the registry of the operating system but will not solve the problem of one student using two or more IDs if Decaf is used across multiple computers by the same student.

Additionally, it will still be possible with some effort to remove an already assigned ID and obtain a new one.

Related to the threat just mentioned, students were encouraged but could not be forced to only use Decaf. A student could chose to use another environment, or use Decaf and another environment, although the lecturer noted very little evidence of this.

A minor issue is that Decaf does not enhance three of the top 15 CEMs: 39 *'else' without 'if'*, 53 *illegal start of type*, and 5 *'.class' expected*. This however did provide to be an interesting 'self-contained' control case which spanned both groups. As both groups experienced the same raw Java CEMs in these cases, it would be expected that there would be little variation in their frequencies. Indeed for one of these (39) Decaf had a slightly higher frequency, and for CEMs 39 and 5, the frequencies were almost equivalent. Additionally, CEM 5 provided an opportunity to briefly explore the possibility of a 'knock-on' effect of ECEMs.

Further discussion is warranted on the ECEMs themselves. Firstly the wording and therefore the 'helpfulness' of the ECEMs is difficult to directly measure. On this topic, future work based on that of Marceau, Fisler, & Krishnamurthi (2011a) is mentioned in Section 5. Secondly, the original Java CEM is presented unaltered, alongside an ESEM (if one is generated) to the intervention group. No examples or other information that could lead to validity issues is presented unlike some other studies. However this could possibly lead to confusion by students being presented with two versions of a compiler error message for a single student error. However, the results of survey questions (particularly open-ended responses) did not show any evidence of this.

The Decaf software was designed before the publication of Denny, Luxton-Reilly and Carpenter (2014), and shares with their research a threat to validity in that the control students were presented with more than one CEM, allowing some students to possibly correct more than one error simultaneously, or potentially confusing other students. This design was an effect of the design decision of not to interfere with the standard Java CEM presentation in any way for the control group.

Looking forward, steps can be taken to eliminate or at least mitigate most of these threats.

This study acknowledges the following limitations and delimitations:

Limitations:

- Java only, as it is the normal language of instruction for this module
- Two groups due to time restrictions
- Approximately 200 students

Delimitations:

- Only compile-time errors were investigated. Decaf does provide ECEMs for some runtime CEMs however - see Appendix H.
- Only 30 CEMs were enhanced. There are several hundred Java CEMs, but many only arise from the use of language keywords and constructs that are not taught in year 1 programming, and therefore are unlikely to occur, except by accident. See Table 3.4 for the 30 CEMs enhanced by Decaf.
- The study period was limited to minimise impact on students.

4.6 Summary Discussion

This section began showing that the control group had an 'error profile' similar to those in other studies on the Java programming language, and possibly with languages other than Java. It then moved through 'hard data' on errors, through individual errors, to errors per student, through group behaviour and then looking at individual students. This journey ended (before discussing threats to validity) in presenting survey findings, bringing us in close contact with the students themselves, in their words, describing experiences and opinions first-hand.

It has been demonstrated that Decaf may have an active role in reducing the number of errors, particularly high frequency errors. The number of errors per student was reduced, and the most frequent individual CEMs (representing the majority of all CEMs generated) were reduced. The number of repeated errors – a reliable metric to identify struggling students was reduced, as well as the number of repeated error strings. The students in the intervention group were shown to 'behave' more similarly to each other – as a more cohesive group – with fewer and less distant outliers, and fewer indications of struggling students when compared to the control group. Students investigated in individual vignettes showed interesting insight into their activity, behaviour, struggles and successes and corroborated findings in previous sections. Finally survey results were presented, hearing from students in their own words, describing a positive experience with Decaf, its role in enhancing compiler error messages, and in learning computer programming.

Chapter 5 Conclusions

"Decaf is a user-friendly software, and by using it we get less errors"
"It makes programming a little easier by showing you the errors"
"It is an amazing way to learn"

– Anonymous students from the intervention group in this study

Computer programming is an expected outcome of almost all computing degrees in higher education (McCracken, et al., 2001) and a core competency for many IT roles (Orsini, 2013), even outside those roles for which it is a primary focus. Learning to program is one of the most significant challenges students face and has been shown to be a barrier to successful outcomes (Stefik & Siebert, 2013), linked to poor student performance and the high attrition rates of many computing programmes in Ireland and globally.

There are many difficulties faced by students learning to program, but few, if any are as persistent and universally experienced as compiler error messages (CEMs). Difficulties students have with CEMs have been present for at least four decades (which in computing is an aeon) and occur with almost all programming languages. They are extremely important as the student's primary source of information on their work, providing instant feedback intended to help students locate, diagnose and correct their own errors, often made just seconds before. Unfortunately they often come in varying mixtures of terse, confusing, too numerous, misleading, and sometimes (apparently) wrong, and are known to be a source of frustration, discouragement and in some cases disdain.

This thesis presented the results of an in-depth empirical investigation on the effects of a Java editor called Decaf, specifically written for this research. Similar to a handful of efforts before, Decaf enhances CEMs (ECEMs), presenting them in what is hopefully a more understandable and usable form. Although only a few similar systems exist, even fewer in-depth empirical studies have been carried out on their effectiveness (Denny, Luxton-Reilly, & Carpenter, 2014), a motivating factor for this research in addition to those above.

The aim of this research was to investigate the question:

Two groups were investigated during the first half of their semester 1 CS1 module, a control group experiencing standard Java CEMs and an intervention group experiencing ECEMs. Each group consisted of approximately 100 students and together they generated nearly 50,000 errors. The control group was shown to have an error distribution very similar to several other studies on Java and other languages, providing a 'baseline' and grounds for generalisation. It was found that the overall number of errors was significantly reduced for the intervention group. Perhaps more importantly, the number of errors per student was also lower, particularly for high frequency errors. Eight CEMs were identified accounting for 43.2% of all errors and all enhanced by Decaf, which had a statistically significant reduced number of errors per student. These eight errors are amongst the most commonly encountered by students in several other studies. The number of repeated errors – a key metric in identifying struggling students was also reduced in addition to the number of repeated error strings.

The data was also analysed from a group perspective, finding that the intervention group had a more homogenous error profile with fewer signs of struggling students. Investigating individual students through vignettes gave an insight into factors that may contribute to students falling far from the centres of their groups, and most likely struggling. These factors include: a high number of errors relative to other students; a high ratio of repeated errors to total errors; and an error profile that deviates more from the expected distribution of errors.

The student experience was explored in their own words and opinions garnered through surveys in addition to lecturer interviews revealing a positive experience with Decaf and ECEMs. This completed a journey from 'hard data' on errors, to errors per student, through group behaviour, followed by individual student profiles, and ending with the lecturer's and the students' first-hand experiences. Perhaps most importantly this revealed that students had a positive learning experience with Decaf and the enhanced error messages it provides.

Implications for Practice

Given that the problems CEMs present to students have been present for over half of the history of high-level computer programming, some predictions are not fool-hardy.

First and foremost, Java will most likely be replaced as the novice language of choice with Python as a top contender. Although this will bring change, the fact remains that several popular novice teaching languages have come and gone over more than four decades, but difficulties presented by CEMs have persisted. This makes it seem unlikely that the problems students encounter with CEMs will be alleviated in the short-term by a language change alone. Secondly, the manner in which data about the problem is gathered will continue to change. Error detection and aggregation is getting increasingly sophisticated. The Blackbox dataset introduced by Brown, Kölling McCall & Utting (2014) contains millions of errors, and has already been used to analyse 37 million of them, from hundreds of thousands of students over at least several hundred institutions (Altadmri & Brown, 2015).

A solution, if there ever is one, will come first from one of three likely sources. The first is language designers themselves, through languages which by nature are less prone to errors rooted in complex syntax and semantics. The second is compiler designers, who have the possibility of discovering and deciphering error causes differently and presenting more useful CEMs to programmers so they can rectify them more effectively. An example is Eclipse which has its own Java compiler and its own CEMs which are arguably better than those of the Java JDK. The third are designers of editors and environments such as Decaf – tools which interpret and 'fix' CEMs, most likely through enhancement. I used the word fix intentionally – I do not believe that such environments will be the final solution (at least in isolation) as they are just that – a fix. Instead I believe that the ultimate solution will be a combination of efforts from language, compiler and editor designers in concert. However, existing languages already exist and already have their flaws. These languages are immensely popular, running the software that the modern world depends on. In addition, there will most likely always be languages with CEMs more notorious than others, and therefore a likely need for enhanced CEMs.

Future Research Directions

Directions for future work follow two directions. The first is further into the data already gathered by applying the rubric of (Marceau, Fisler, & Krishnamurthi, 2011a) designed to identify specific error messages that are problematic for students. Although this research identified specific error messages, these were based on frequency, not analysing the actual issues students encountered when they committed particular errors.

In addition, further analysis on repeated error data is possible. The second direction is improving Decaf and gathering more data. Ultimately, it is envisioned that Decaf will be available for download and use by any student learning computer programming at any institution which teaches Java to beginners. In addition a web-based editor is envisioned, requiring no download or installation of software. Both will provide scope for future study by including more institutions, greater student diversity, and a greater overall number of participants.

It is unreasonable to think that enhancing compiler error messages will completely alleviate the problems students have with them. However it has been shown that Decaf reduced student errors, reduced indications of struggling students, and provided a positive learning experience. It is hoped that this experience can be shared and help more students, providing assistance in one of the many hurdles computer programming students face in learning an extremely important skill.

References

Including those cited in appendices

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. *ACM SIGCSE Bulletin, 37*(3), 84-88.

Allen, E., Cartwright, R., & Stoler, B. (2002). DrJava: A lightweight pedagogic environment for Java. *ACM SIGCSE Bulletin, 34*(1), 137-141.

Altadmri, A., & Brown, N. (2015). 37 million compilations: Investigating novice programming mistakes in large-scale student data. *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 522-527). Kansas City: ACM.

Barnes, D. J., & Kölling, M. (2011). *Objects first with Java: A practical introduction using BlueJ* (5th ed.). London: Prentice Hall.

Beaubouef, T., & Mason, J. (2005). Why the high attrition rate for computer science students: some thoughts and observations. *ACM SIGCSE Bulletin, 37*(2), 103-106.

Ben-Ari, M. (1998). Constructivism in computer science education. *ACM SIGCSE Bulletin, 30*(1), 257-261.

Ben-Ari, M. (2007). *Compile and runtime errors in Java.* Rehovot, Israel: Department of Science Teaching, Weizmann Institute of Science.

Bergin, J., Agarwal, A., & Agarwal, K. (2003). Some deficiencies of C++ in teaching CS1 and CS2. *ACM SIGPLAN Notices, 38*(6), 9-13.

Bergin, S., & Reilly, R. (2005). Programming: factors that influence success. *ACM SIGCSE Bulletin, 37*(1), 411-415.

Brown, N. C., & Altadmri, A. (2014). Investigating novice programming mistakes: educator beliefs vs student data. *Proceedings of the tenth annual conference on International computing education research - ICER '14* (pp. 43-50). ACM.

Brown, N. C., Kölling, M., McCall, D., & Utting, I. (2014). Blackbox: A large scale repository of novice programmers' activity. *The 45th SIGCSE technical symposium on computer science education (SIGCSE* (pp. 223-228). ACM.

Brown, P. (1983). Error messages: the neglected area of the man/machine interface. *Communications of the ACM, 26*(4), 246-249.

Bruce, K. B. (2004). Controversy on How to Teach CS1: A Discussion on the SIGCSE-members Mailing List. *SIGCSE Bulletin (inroads), 36*(4), 29-34.

Burgess, M. (1999). *C programming tutorial 4th edition (K&R version).* Oslo College, Facutly of Engineering.

Carter, E., & Blank, G. D. (2013). A tutoring system for debugging: status report. *Journal of Computing Sciences in Colleges, 28*(3), 46-52.

Caspersen, M. E., & Bennedsen, J. (2007). Instructional design of a programming course - A learning theoretic approach. *Proceedings of the third international workshop on Computing education research* (pp. 111-122). ACM.

Cass, S. (2015, July 20). *The 2015 top ten programming languages.* Retrieved from IEEE Spectrum: http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages

Chamillard, A., & Hobart Jr. , W. C. (1997). Transitioning to Ada in an introductory course for non-majors. *Proceedings of the conference on TRI-Ada '97* (pp. 37-40). ACM.

Chan-Mow, I. (2012). Analyses of student programming errors in Java programming courses. *Journal of Emerging Trends in Computing and Information Sciences*, 739-745.

Chatley, R., & Timbul, T. (2005). KenyaEclipse: learning to program in eclipse. *ACM SIGSOFT Software Engineering Notes, 30*(5), 245-248.

Check, J., & Schutt, R. K. (2011). *Reserch methods in education.* Thousand Oaks, Calif: SAGE Publications.

Corney, M., Teague, D., Ahadi, A., & Lister, R. (2012). Some empirical results for neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. *Proceedings of Conferences in Research and Practice in Information Technology (CRPIT)* (pp. 77-86). Australian Computer Society, Inc.

Coull, N. J. (2008). SNOOPIE: Development of a learning support tool for novice programmers within a conceptual framework. *PhD Thesis*. University of St. Andrews.

Coull, N. J., & Duncan, I. M. (2011). Emergent requirements for supporting introductory programming. *Innovation in Teaching and Learning in Information and Computer Sciences, 10*(1), 78-85.

Davies, S., Polack-Wahl, J. A., & Anewalt, K. (2011). A snapshot of current practices in teaching the introductory programming sequence. *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 625-630). ACM.

Debuse, J., & Lawley, M. (2012). The learning and productivity benefits to student programmers from real-world development environments. *Information Systems Education, 10*(5), 61 - 81.

Denny, P., Luxton-Reilly, A., & Carpenter, D. (2014). Enhancing syntax error messages appears ineffectual. *Proceedings of the 2014 conference on Innovation & technology in computer science education* (pp. 273-278). ACM.

Denny, P., Luxton-Reilly, A., & Tempero, E. (2012). All syntax errors are not equal. *ITiCSE 12* (pp. 75-80). ACM.

Denny, P., Luxton-Reilly, A., Tempero, E., & Hendrickx, J. (2011). CodeWrite: supporting student-driven practice of java. *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 471-476). ACM.

Do people in non-English-speaking countries code in English? (2013). Retrieved July 17, 2015, from StackExchange: http://programmers.stackexchange.com/questions/1483/do-people-in-non-english-speaking-countries-code-in-english

Dy, T., & Rodrigo, M. (2010). A detector for non-literal Java errors. *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (pp. 118-122). ACM.

Etheridge, J. (2004). CMeRun: program logic debugging courseware for CS1/CS2 students. *ACM SIGCSE Bulletin, 36*(1), 22-25.

Falkner, N. (2014, June 26). *ITiCSE 2014, Day 3, Final Session, "CS Ed Research", #ITiCSE2014 #ITiCSE*. Retrieved June 18, 2015, from Nick Falkner Learning and Teaching in ICT at The University of Adelaide and across Australia: http://nickfalkner.com/2014/06/26/iticse-2014-day-3-final-session-cs-ed-research-iticse2014-iticse/

Farragher, L., & Dobson, S. (2000). Java Decaffeinated: experiences building a programming language from components.

Fenichel, R. R., Weizenbaum, J., & Yochelson, J. C. (1970). A program to teach programming. *Communications of the ACM, 13*(3), 141-146.

Fenwick, Jr., J. B., Norris, C., Barry, F. E., Rountree, J., Spicer, C. J., & Cheek, S. D. (2009). Another look at the behaviors of novice programmers. *Proceedings of the 40th ACM technical symposium on computer science education* (pp. 296-300). ACM.

Flowers, T., Carver, C. A., & Jackson, J. (2004). Empowering students and building confidence in novice programmers through Gauntlet. *Frontiers in Education, 2004. FIE 2004. 34th Annual* (pp. T3H-10). IEEE.

Graham, S. L., & Rhodes, S. P. (1973). Practical syntactic error recovery in compilers. *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (pp. 52-58). ACM.

Gray, D. E. (2013). *Doing research in the real world.* London: SAGE Publications.

Gries, D. (1974). What should we teach in an introductory programming course? *Proceedings of the fourth SIGCSE technical symposium on computer science education* (pp. 81-89). ACM.

Guo, P. (2014, July 7). *Python is now the most popular introductory teaching language at top U.S. universities*. Retrieved March 20, 2015, from Communications of the ACM: http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext

Guzdial, M. (2011, January 24). *Predictions on future CS1 languages.* Retrieved June 21, 1995, from Computing Education Blog: https://computinged.wordpress.com/2011/01/24/predictions-on-future-cs1-languages/

Guzdial, M. (2014, July 29). *Enhancing syntax error messages appears ineffectual — if you enhance the error messages poorly*. Retrieved June 19, 2015, from Computing Education Blog: https://computinged.wordpress.com/2014/07/29/enhancing-syntax-error-messages-appears-ineffectual-if-you-enhance-the-error-messages-poorly/

Guzdial, M. (2014, March 27). *Programming languages are the most powerful, and least usable and learnable user interfaces*. Retrieved June 21, 2015, from Communications of the ACM: http://cacm.acm.org/blogs/blog-cacm/173328-

programming-languages-are-the-most-powerful-and-least-usable-and-learnable-user-interfaces/fulltext

Hadjerrouit, S. (1998). Java as first programming language: a critical evaluation. *ACM SIGCSE Bulletin, 30*(2), 43-47.

Hanselman, S. (2008, November 8). *Do you have to know english to be a programmer?* Retrieved July 24, 2015, from www.hanselman.com: http://www.hanselman.com/blog/DoYouHaveToKnowEnglishToBeAProgrammer.aspx

Hartmann, B., MacDougall, D., Brandt, J., & Klemmer, S. R. (2010). What would other programmers do? Suggesting solutions to error messages. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 1019-1028). ACM.

Heeren, B., Leijen, D., & van IJzendoorn, A. (2003). Helium, for learning Haskell. *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell* (pp. 62-71). ACM.

Hertz, M. (2010). What do CS1 and CS2 mean? Investigating differences in the early courses. *Proceedings of the 41st ACM technical symposium on Computer science education* (pp. 199-203). ACM.

Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. *ACM SIGCSE Bulletin, 35*(1), 153-156.

Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. *Proceedings of the 34th SIGCSE technical symposium on computer science education* (pp. 153-156). ACM.

Hsia, J. I., Simpson, E., Smith, D., & Cartwright, D. (2005). Taming Java for the classroom. *ACM SIGCSE Bulletin*, 327-331.

Jackson, J., Cobb, M., & Carver, C. (2005). Identifying top Java errors for novice programmers. *Frontiers in Education, 2005. FIE'05. Proceedings 35th Annual Conference* (pp. T4C24 - T4C27). IEEE.

Jadud, M. C. (2005). A first look at novice compilation behaviour using BlueJ. *Computer Science Education, 15*(1), 25-40.

Jadud, M. C. (2006). An exploration of novice compilation behaviour in BlueJ. *PhD Thesis*. University of Kent. Retrieved from http://www.cs.kent.ac.uk/pubs/2007/2615/

Johnson, W. L., & Soloway, E. (1984). PROUST: Knowledge-based program understanding. *ICSE '84 Proceedings of the 7th international conference on Software engineering* (pp. 369-380). IEEE.

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR), 37*(2), 83-137.

Ko, A. (2014, March 25). *Programming languages are the least usable, but most powerful human-computer interfaces ever invented*. Retrieved June 21, 2015, from Bits and behaviour: http://blogs.uw.edu/ajko/2014/03/25/programming-languages-are-the-least-usable-but-most-powerful-human-computer-interfaces-ever-invented/

Kölling, M. (1999, February). The design of an object-oriented environment and language for teaching. *PhD Thesis*. Basser Department of Computer Science, University of Sydney. Retrieved from http://www.cs.kent.ac.uk/pubs/1999/2171/

Kölling, M., & Rosenburg, J. (1996a). An object-oriented program development environment for the first programming course. *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education* (pp. 83-87). ACM.

Kölling, M., & Rosenburg, J. (1996b). Blue - A Language for Teaching Object-Oriented Programming. *Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education*, (pp. 290-294).

Kölling, M., Koch, B., & Rosenburg, J. (1995). Requirements for a first year object-oriented teaching language. *SIGCSE Bulletin, 27*(1), 173-177.

Koster, C. (1973). Error Reporting, Error Treatment, and Error Correction in Algol Translation—Part 1. *GI. Gesellschaft für Informatik eV 2. Jahrestagung* (pp. 179-187). Springer Berlin Heidelberg.

Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM, 50*(4), 36-42.

Kummerfeld, S. K., & Kay, J. (2003). The neglected battle fields of syntax errors. *Proceedings of the fifth Australasian conference on Computing education-Volume 20* (pp. 105-111). Australian Computer Society.

Lang, B. (2002). Teaching new programmers: a Java tool set as a student teaching aid. *Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002* (pp. 95-100). National University of Ireland.

Li, X., & Prasad, C. (2005). Effectively Teaching Coding Standards in Programming. *Proceedings of the 6th conference on information technology education* (pp. 239-244). ACM.

Lister, R. (2011). Concrete and other neo-Piagetian forms of reasoning in the novice programmer. *Proceedings of the Thirteenth Australasian Computing Education Conference-Volume 114* (pp. 9-18). Australian Computer Society, Inc.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., . . . Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin, 36*(4), 119-150.

Litecky, C. R., & Davis, G. B. (1976). A study of errors, error-proneness, and error diagnosis in Cobol. *Communications of the ACM, 19*(1), 33-38.

Marceau, G., Fisler, K., & Krishnamurthi, S. (2011a). Measuring the effectiveness of error messages designed for novice programmers. *Proceedings of the 42nd ACM technical symposium on computer science education* (pp. 499-504). ACM.

Marceau, G., Fisler, K., & Krishnamurthi, S. (2011b). Mind your language: on novices' interactions with error messages. *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software* (pp. 3-18). ACM.

Matthíasdóttir, Á., & Geirsson, H. J. (2011). The novice problem in computer science. *Proceedings of the 12th International Conference on Computer Systems and Technologies* (pp. 570-576). ACM.

Matuszek, D. (n.d.). *Errors.* Retrieved from http://www.cis.upenn.edu/~matuszek/General/JavaSyntax/errors.html

McCall, D., & Kölling, M. (2014). Meaningful categorisation of novice programmer errors. *Proceedings of the 2014 International Conference on Frontiers in Education: Computer Science and Computer Engineering* (pp. 2589-2596). CSREA Press.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., . . . Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin, 33*(4), 125-180.

Mertens, D. M. (2014). *Research and evaluation in education and psychology: Integrating diversity with quantitative, qualitative, and mixed methods.* London: Sage Publications.

Mooney, O., Patterson, V., O'Connor, M., & Chantler, A. (2010). *A study of progression in Irish higher education.* Dublin: Higher Education Authority (Ireland).

Moore, S. (2005). An intelligent interactive online tutor for computer languages. *25th Annual International Conference of the British Computer Society's Specialist Group on Artificial Intelligence (SGAI).*

Motil, J., & Epstein, D. (n.d.). *JJ: a language designed for beginners.* Retrieved June 18, 2015, from http://www.ecs.csun.edu/~jmotil/TeachingWithJJ.pdf

Moulton, P. G., & Muller, M. E. (1967). DITRAN—a compiler emphasizing diagnostics. *Communications of the ACM, 10*(1), 45-52.

Murphy, C., Kim, E., Kaiser, G., & Cannon, A. (2008). Backstop: a tool for debugging runtime errors. *ACM SIGCSE Bulletin, 40*(1), 173-177.

Murray, K. A., Kölling, M., Schaller, N. C., Heines, J. M., Moore, T., & Trono, J. A. (2003). Experiences with IDEs and Java teaching:What works and what doesn't. *ACM SIGCSE Bulletin, 35*(3), 215-216.

Nielsen, J. (1994). Heuristic evaluation. In J. Nielsen, & R. L. Mack, *Usability Inspection Methods* (pp. 25-62). New York: John Wiley & Sons.

Nienaltowski, M.-H., Pedroni, M., & Meyer, B. (2008). Compiler error messages: what can help novices? *ACM SIGCSE Bulletin, 40*(1), 168-172.

Odekirk-Hash, E., & Zachary, J. L. (2001). Automated feedback on programs means students need less help from teachers. *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education* (pp. 55-59). ACM.

Oracle Corporation. (2015). *Java SE specifications.* Retrieved March 20, 2015, from Oracle Help Center: http://docs.oracle.com/javase/specs/

Orsini, L. (2013, May 31). *Why programming is the core skill of the 21st century*. Retrieved from readwrite: http://readwrite.com/2013/05/31/programming-core-skill-21st-century

Pane, J. F., & Myers, B. A. (1996). *Usability issues in the design of novice programming systems.* Technical Report, Carnegie Mellon University, School of Computer Science. Retrieved April 4, 2015, from http://www.cs.cmu.edu/~pane/cmu-cs-96-132.html

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., . . . Paterson, J. (2007). A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin, 39*(4), 204-223.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. *Journal of Educational Computing Research, 2*(1), 37-55.

Peters, A., & Pears, A. (2012). Students' experiences and attitudes towards learning computer science. *Frontiers in Education Conference (FIE)* (pp. 1-6). IEEE.

Phillips, D., & Burbules, N. C. (2000). *Postpositivism and educational research.* Lanham, Maryland: Rowman & Littlefield Publishers.

Piteira, M., & Costa, C. (2011). Innovate in your program computer class: an approach based on a serious game. *Proceedings of the 2011 Workshop on Open Source and Design of Communicationv* (pp. 49-54). ACM.

Porter, L., Guzdial, M., McDowell, C., & Simon, B. (2013). Success in introductory programming: what works? *Communications of the ACM, 58*(8), 34-36.

Programming Language Popularity. (2013, October 25). Retrieved from langpop.com: http://www.langpop.com

Reis, C., & Cartwright, R. (2003). A friendly face for eclipse. *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange* (pp. 25-29). ACM.

Reis, C., & Catrwright, R. (2004). Taming a professional IDE for the classroom. *ACM SIGCSE Bulletin, 36*(1), 156-160.

Rigby, P., & Thompson, S. (2003). Study of novice programmers using Eclipse and Gild. *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange* (pp. 105-109). ACM.

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education, 13*(2), 137-172.

Rodrigo, M. T., Baker, R. S., Jadud, M. C., Amarra, A. C., Dy, T., Espejo-Lahoz, M. V., . . . Tabano, E. S. (2009). Affective and behavioral predictors of novice prgrammer behaviour. *ACM SIGCSE Bulletin, 41*(3), 156-160.

Rogerson, C., & Scott, E. (2010). The fear factor: how it affects students learning to program in a tertiary environment. *Journal of Information Technology Education: Research, 9*(1), 147-171.

Rosenburg, J., & Kölling, M. (1997). Testing object-oriented programs: making it simple. *ACM SIGCSE Bulletin, 29*(1), 77-81.

Ryan, A. B. (2006). Post-Positivist Approaches to Research. In M. Anotonesa, *Researching and writing your thesis: A guide for postgraduate students* (pp. 12-26). Maynooth, Ireland: NUI Maynooth.

Schorsch, T. (1995). CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. *ACM SIGCSE Bulletin, 27*(1), 168-172.

Shlens, J. (2003, March 25). *A tutorial on principal component analysis: derivation, discussion, and singular value decomposition.* Retrieved August 14, 2015, from https://www.cs.princeton.edu/picasso/mats/PCA-Tutorial-Intuition_jp.pdf

Shneiderman, B. (1982). Designing computer system messages. *Communications of the ACM, 25*(9), 610-611.

Shuhidan, S., Hamilton, M., & D'Souza, D. (2009). A taxonomic study of novice programming summative assessment. *Proceedings of the Eleventh Australasian Conference on Computing Education. 95,* pp. 147-156. Australian Computer Society, Inc.

Siegfried, R. M., Greco, D., Miceli, N., & Siegfried, J. (2012). Whatever happened to Richard Reid's list of first programming languages? *Information Systems Education Journal, 10*(4), 24-30.

Sigfried, R. M., & Chays, D. (2008). Will there ever be consensus on CS1? *Proceedings of Frontiers in Education: Computer Science and Computer Engineering* (pp. 18-23). CSREA Press.

Sloan, R. H., & Troy, P. (2008). CS 0.5: A better approach to introductory computer science for majors. *ACM SIGCSE Bulletin, 40*(1), 271-275.

Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct. *Communications of The ACM, 29*(7), 624-632.

Stake, R. E. (1998). Case Studies. In N. K. Denzin, & Y. S. Lincoln (Eds.), *Strategies of qualitative inquiry (Handbook of qualitative research paperback edition)* (Vol. 2). Thousand Oaks, California, USA: SAGE Publications.

Stefik, A., & Siebert, S. (2013). An empirical investigation into programming language syntax. *ACM Transactions on Computing Education, 13*(4), 19:1-19:40.

Storey, M.-A., Damian, D., Michaud, J., Myers, D., Mindel, M., German, D., . . . Hargreaves, E. (2003). Improving the usability of Eclipse for novice programmers. *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange* (pp. 35-39). ACM.

Tabano, E. S., Rodrigo, M. T., & Jadud, M. C. (2011). Predicting at-risk novice Java programmers through the analysis of online protocols. *Proceedings of the seventh international workshop on Computing education research* (pp. 85-92). ACM.

Thomas, L., Ratcliffe, M., Woodbury, J., & Jarman, E. (2002). Learning styles and performance in the introductory programming sequence. *ACM SIGCSE Bulletin, 34*(1), 33-37.

Thompson, S. M. (2004). An exploratory study of novice programming experiences and errors. *MSc Thesis*. University of Victoria.

TIOBE Software. (2015, July). *TIOBE Index for July 2015*. Retrieved April 1, 2015, from TIOBE Software: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Toomey, W. (n.d.). *Quantifying the incidence of novice programmers' errors.* Gold Coast, Australia: School of IT, Bond University.

Toomey, W., & Gjengset, J. (2011, July). *Arjen: A tool to identify common programming errors*. Retrieved from http://minnie.tuhs.org/Programs/Arjen/

Traver, V. J. (2010). On compiler error messages: What they say and what they mean. *Advances in Human-Computer Interaction, 2010*, 1-26.

Trochim, W. M. (2006, October 20). *Positivism & post positivism*. Retrieved August 27, 2014, from Research methods knowledge base: http://www.socialresearchmethods.net/kb/positvsm.php

van Tonder, M., Naudé, K., & Cilliers, C. (2008). Jenuity: a lightweight development environment for intermediate level programming courses. *ACM SIGCSE Bulletin, 40*(3), 58-62.

Veerasamy, A. K., & Shillabeer, A. (2014). Teaching English based programming courses to English language learners/non-native speakers of English. *International Proceedings of Economics Development and Research, 70*(4), 17-22.

Vogts, D., Calitz, A. A., & Greyling, J. H. (2010). The effects of professional and pedagogical program development environments on novice programmer perceptions. *South African Computer Journal, 45*, 53-58.

Watson, C., & Li, F. W. (2014). Failure rates in introductory programming revisited. *Proceedings of the 2014 conference on Innovation & technology in computer science education* (pp. 39-44). ACM.

Weildt, T. (2014, October 27). *Java SE 8 on Java.com*. Retrieved August 20, 2015, from www.java.com: https://blogs.oracle.com/java/entry/java_se_8_on_java

Wexelblat, R. L. (1976). Maxims for malfeasant designers, or how to design languages to make programming as difficult as possible. *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)* (pp. 331-336). IEEE Computer Society Press.

Yadin, A. (2011). Reducing the dropout rate in an introductory programming course. *ACM Inroads, 2*(4), 71-76.

Ziring, N. (2001, July 1). *Novice Java programmers' favorite mistakes*. Retrieved January 13, 2015, from http://users.erols.com/ziring/java-npm.html

Zuber-Skerritt, O. (1992). *Professional development in higher education.* London: Kogan Page Limited.

Appendix A Programming for Non-native English Speakers

It should be noted that in general non-native English speakers program and learn in English, as almost all programming languages are designed using English (Veerasamy & Shillabeer, 2014), as are most sources of documentation (Li & Prasad, 2005). Attempts to develop programming languages in natural languages other than English have been few, and have not gained popularity or use at university level teaching (Veerasamy & Shillabeer, 2014). Some non-native English speakers choose to use their native language for comments and variable, method and function names; for a discussion on this, see (Do people in non-English-speaking countries code in English?, 2013).

The challenges faced by non-native English speakers when learning how to program are poorly represented in the literature, and there is relatively little research done on students who commit syntax errors or use improper keywords and judgment errors due to their lack of English proficiency (Veerasamy & Shillabeer, 2014). In a small study, Li and Prasad (2005) found that none of the native English speakers found it difficult to follow coding standards while 25% of non-native English speakers found the topic difficult. They also found that native English speakers prefer examples and practice much more than non-native English speakers while non-native English speakers prefer lectures more than native English speakers; however they felt that this is more of a cultural issue than a language issue, and warrants further research. They concluded that different strategies may be required for teaching native and non-native English speakers programming, but again this requires further research. A google scholar search carried out by the author with the keywords +“computer programming” +“non-native English” showed surprisingly few results (262), only a few of which actually involved non-native English speakers studying or doing computer programming. Adding +syntax in the query reduced this to 92 results, Compared to about 757,000 for “computer programming” and about 25,400 for “non-native English”. The topic of whether one needs to know English in order to be a successful programmer is discussed in communities of programmers however, in one case drawing 128 comments in the course of three months (Hanselman, 2008). Another discussion has been ongoing for four years, with 108 comments (Do people in non-English-speaking countries code in English?, 2013).

Appendix B Most Frequent Java Errors from Eleven Studies

The majority of these studies report compiler error messages (*). Some report student committed errors which *generate* compiler error messages (**). At times, some are ambiguous.

<table>
<tr><td colspan="2">(Hristova, Misra, Rutter, & Mercuri, 2003)**</td></tr>
<tr><td>1. Using = instead of == or vice-versa
2. Mismatching, miscounting and/or misuse of { }, [ ], ( ), “ ”, and ‘ ’
3. Wrong separators in for loops
4. An if followed by a bracket instead of by a parenthesis
5. Using keywords as method names or variable names
6^. Invoking methods with wrong arguments
7. Forgetting parentheses after method call
8. Incorrect semicolon at the end of a method header</td><td>9. Leaving a space after a period when calling a specific method
10. >= and =
11*. Invoking class method on object
12. Improper casting
13*. Flow reaches end of non-void method
14. Methods with parameters: confusion between declaring parameters of a method and passing parameters in a method invocation
15. Incompatibility between the declared return type of a method and in its invocation
16*. Class declared abstract because of missing function</td></tr>
<tr><td colspan="2">* Have a one-to-one mapping with CEMs. The others do not have a one-to-one mapping with student errors (Altadmri & Brown, 2015).</td></tr>
<tr><td colspan="2">(Flowers, Carver, & Jackson, 2004)**</td></tr>
<tr><td>1. Mismatching curly braces
2. Mismatching quotations
3. Misplaces semicolon
4. Improper file name
5. Not initializing a variable before attempting to use it</td><td>6. Mismatching parenthesis
7. Missing semicolon
8. Misspelling printLine method
9. Package does not exist</td></tr>
<tr><td colspan="2">(Toomey, n.d.)</td></tr>
<tr><td>1. Assignment in if statement
2. Use of comparison after Boolean operator
3. Use of bitwise operators
4. Cannot find a certain identifier
5. Please use braces not parentheses
6. cannot treat char Like a String
7. Else without a Matching if
8. Empty statement
9. Empty statement after if
10. System.exit() needs a value
11. Missing identifier
12. Probable code in wrong place or missing braces /
parentheses
13. Probable imbalance with braces
14. Incomparable types</td><td>16. Missing left brace
17 Possible loss of precision
18. Malformed for loop
19. Possible misspelt word or command
20: Not a statement
21. Package does not exist
22. Not enough closing braces
23. Right parenthesis expected
24. Missing ;
25. Checking for String (in)equality
26. Missing “ or “ in String literal
27. Unrequired extra type keyword used
28. Duplicate variable
29. Cannot use something which gives ‘void’ in an expression</td></tr>
<tr><td colspan="2">(Thompson, 2004)</td></tr>
<tr><td>1. Undefined name</td><td>6. Undefined type</td></tr>
</table>

2. Type mismatch 3. Undefined method 4. parsing error insert to complete 5. Should return value	7. Parsing error delete token 8. Package is not expected package 9. Undefined constructor 10. Parameter mismatch
(Jackson, Cobb, & Carver, 2005)	
1. Cannot resolve symbol 2. ; expected 3. Illegal start of expression 4. class or interface expected 5. <identifier> expected 6.) expected 7. Incompatible types 8. *int* 9. Not a statement 10. } expected	11. *class FinalProject* 12. Illegal start of type 13. *java.lang.string* 14. Invalid method declaration; return type required 15. *boolean* 16. *else* without *if* 17. { expected 18. *double* 19. (expected 20. possible loss of precision
(Jadud, 2006)*****	
1. Unknown variable 2. Bracket expected 3. Unknown method 4. Semicolon expected 5. Illegal start of expression	6. Unknown class 7. Incompatible types 8. Method application error 9. Private access violation 10. Missing return
(Dy & Rodrigo, 2010)*****	
1. Unknown variable 2. ‘;’ expected 3. ‘[‘, ‘]’, ‘(‘, ‘)’, “, ” expected 4. unknown method 5. incompatible types	6. missing return statement 7. illegal start of expression 8. unknown class 9. identifier expected 10. class or interface expected
(Tabano, Rodrigo, & Jadud, 2011)*****	
1. cannot find symbol – variable 2. ‘;’ expected 3. ‘(’ or ‘)’ or ‘[‘ or ‘]’ or ‘{‘ or ‘}’ expected 4. missing return statement 5. cannot find symbol – method	6. illegal start of expression 7. incompatible types 8. <identifier> expected 9. class, interface or enum expected 10. cannot find symbol - class
(Chan-Mow, 2012)*****	
1. Variable not found 2. Identifier expected 3. Class not found 4. Mismatched brackets/parenthesis	5. Invalid method declaration 6. Illegal start of type 7. Method not found 8. Expected
(Denny, Luxton-Reilly, & Tempero, All syntax errors are not equal, 2012)*	
1. Cannot resolve identifier 2. Type mismatch 3. Missing ; 4. Token should be deleted 5. Method not returning correct type	6. Missing } 7. Missing) 8. Missing { 9. Using .length as a field 10. Insert “Assignment Operator”
(Brown, Kölling, McCall, & Utting, 2014)*****	

1. Unknown variable	6. Incompatible types
2. Semicolon expected	7. Illegal start of expression
3. Unknown method	8. Method appliterlication error
4. Bracket expected	9. Identifier expected
5. Unknown class	10. Not a statement

Appendix C Design of the Decaf Editor

There are relatively few tools designed specifically for novice programmers and as such there is not an abundance of information on designing them. Kelleher and Pauch (2005) provided a taxonomy for some existing systems. Decaf's placement in this taxonomy is discussed in Section 3.6.1. The principal design consideration that influenced the design of Decaf was that the Java CEMs could, and should, be improved upon. Michael Kölling, the developer of BlueJ, said of error messages (1999, pp. 145-146):

> *Good error messages make a big difference in the usability of a system for beginners. Often the wording of a message alone can make all the difference between a student being unable to solve a problem without help from someone else and a student being able to quickly understand and remove a small error. The first student might be delayed for hours or days if help is not immediately available (and even in a class with a tutor it may take several minutes for the tutor to be able to provide the needed help).*

At a panel discussion about IDEs for novice Java programmers (Murray, et al., 2003) it was stated that such environments should be simple, stable and affordable. Perhaps these first two requirements seem obvious; however lack of simplicity is arguably the single largest criticism against the use of professional IDEs with novices (Reis & Catrwright, 2004). The open source revolution along with source code repositories and greater collaboration have gone some way towards helping the stability of small projects, and a long way towards making all software more affordable, but the fact of the matter is that pedagogical IDEs are a niche 'market' that must not lose sight of these simple requirements.

Pane & Myers (1996) highlighted simple and useful error messages as one of the requirements of programming environments for novices, something which Nielsen also agreed with: "Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution" (1994, p. 30). Lang (2002) also noted that compilers should not be exempt from user interface considerations, and that good error messages are important, especially for new users and compiler writers should consider their audience when designing error messages.

Nonetheless, there is no definitive list of requirements for a development environment for novices, and a number of curricula and methodologies often require different features in an IDE (Thompson, 2004). In this absence, Decaf was designed by looking at the features of other similar environments. Like in the Gild IDE (Thompson, 2004), and recommended by (Murray, et al., 2003), Decaf features syntax highlighting which colours Java keywords and comments differently to the rest of the code. Also like Gild, line numbers are displayed. Code completion and templates do not feature in Gild or Decaf, as writing code from scratch is important for novices, at least according to some instructors (Thompson, 2004). Also common between Decaf and Gild are that ESEMs are presented in plain English, and kept brief.

BlueJ is the most popular novice IDE, and Decaf borrows from it (and other environments) the fact that only one compiler error at a time is presented to students, allowing them to focus on one error at a time, effectively eliminating cascading errors, unlike Expresso. Although decided upon before the publication of Denny, Luxton-Reilly and Carpenter's 2014 work, presenting one error at a time turned out to be important, as the authors of that study listed the fact that students were presented with two (raw) errors at a time as a threat to validity. This is discussed further in Chapter 5.

In conducting the only recent control/intervention study on enhanced ECEMs, Denny, Luxton-Reilly, & Carpenter (2014) concluded that enhanced error feedback did not have a measurable impact on the study participants' learning. There were several aspects of their study design which supported design decisions that went into the design of Decaf in addition to presenting one error at a time:

1. Decaf captures error information in the most 'natural' or 'live' way possible. Assignments are 'blank-sheet' projects, no skeleton code, method definitions or other structures are provided.
2. Decaf is a 'normal' editor – it is not based on any premise or viewpoint, unlike BlueJ which is based on an 'Objects-First' premise and Code-write which is based on 'drill and practice activities'.
3. Decaf captures the whole learning experience - Decaf logs errors generated during all programming – assignments, practice, examples, lab exercises, etc., not just errors generated during one type of programming activity.

4. The JDK CEM is presented, unaltered, alongside an ESEM (where one is generated) error. No examples or other information that could lead to validity issues is presented.

Along with the above design decisions, the design of Decaf took the clear presentation of ECEMs first and foremost, along with a simple interface, and simple vocabulary as recommended by Marceau, Fisler, & Krishnamurthi (2011a).

It should be noted that the fourth point above was also informed by Coull and Duncan, (2011, p. 80):

> *No support tool that aids program formulation with respect to syntax should promote dependence on that tool to such an extent that the student is not able to progress beyond the remit of that tool. To remove this dependence, and to benefit from support that takes account of the taught context, it is necessary to present the student with both the standard compiler messages and any supplementary error messages enhanced with context-relevant support.*

Lang (2002) noted that some practitioners might feel vaguely uneasy about the wisdom of providing tools such as Decaf to students, possibly thinking that they will spoil the students' academic health and cause problems when the students have to be weaned off them. However Lang provided the following as arguments against these feelings:

1. Compilers should not be exempt from user interface considerations. Most application developers understand the importance of providing good quality error messages, especially for new users. Compiler writers should not expect to be immune from this trend and should consider the audience for error messages and decide whether they are better aimed at the experienced programmer (who probably does not need them), or the novice (who certainly does).
2. Students should be self-sufficient. When faced with a problem, the student should be able to find out what has gone wrong and fix it with minimal intervention from the teacher. Students who solve problems in this way gain valuable experience, learn lessons, and gain confidence. By providing ECEMs, the tools encourage students to solve their problems without having to ask for help. By contrast, a student who is faced with a succession of problems each

of which has to be fixed by a teacher may lose confidence and may even eventually drop out.

3. Students need multiple repetitions of information before it is remembered and understood. Tools such as these are a way of providing these repetitions in a manner which is tailored to the students' immediate needs while saving the teacher's time.

Lang concluded that students will wean themselves off of tools such as Decaf, more out of necessity, rather than stick with them out of dependence. As students gain experience, he felt that students will find the tool's rigidity to be too constraining and so will use it less often, perhaps only when there is a particular problem. In this way, students will naturally and gradually wean themselves off the tools at their own pace.

Finally, Decaf does not include any features that 'do the work for the student'. Such features such as the 'quick fix' option included in KenyaEclipse (Chatley & Timbul, 2005) discourage students to learn by doing, as they may miss the subtlety of some of the changes that are automatically made to their code. Doing so to fix syntax errors is not considered to be helpful, and would contradict the very reason compilers have been producing CEMs for decades.

Finally, Coull and Duncan's 9th and 10th requirements are addressed (2011):

9. *All forms of support may be progressively reduced over the teaching period*

   Decaf is designed to be used during the beginning of a student's programming experience. It has always been envisioned that Decaf users will 'graduate' to other IDEs in the course of their learning.

10. *Use of the tool must be voluntary on the part of the student*

The use of Decaf is completely voluntary. After the study period students were allowed to continue to use Decaf, with an extra menu toggle allowing students to switch the

enhanced error feature on (displaying ESEMs alongside the Java SEMs) or off (displaying only the raw Java SEMs).

Below are screenshots of the software providing a user's perspective;

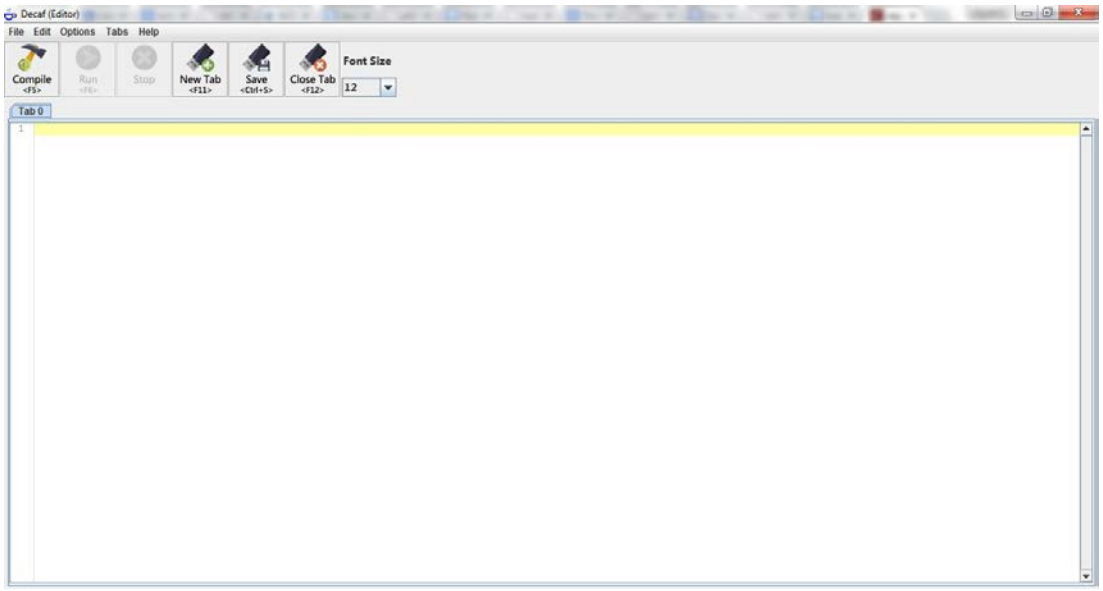

Figure C.1 Decaf after opening, with no active files being edited.

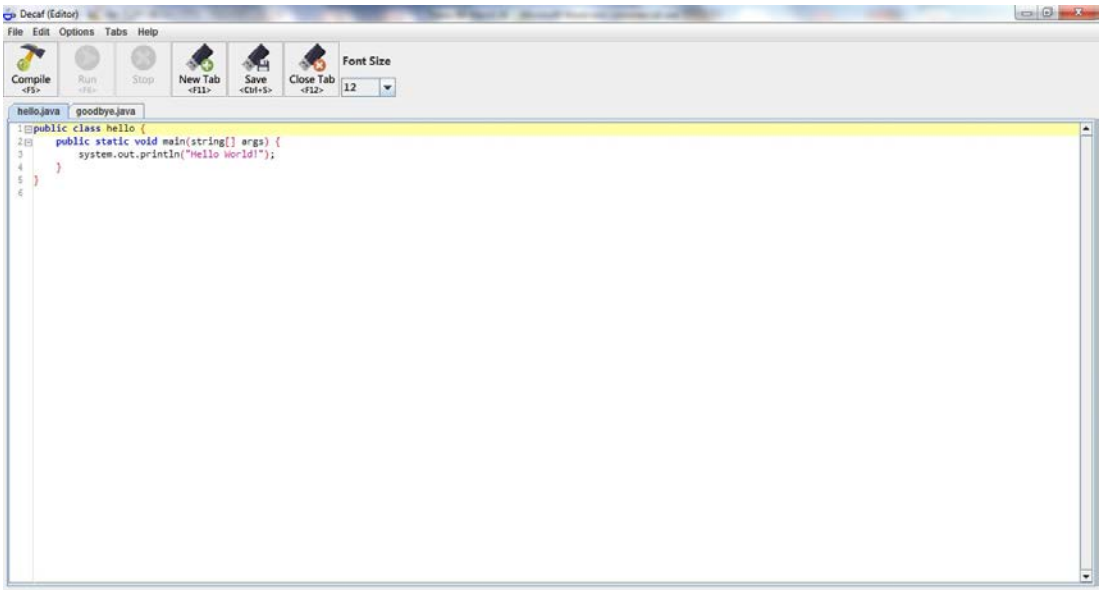

Figure C.2 Decaf with two files open *hello.java* (active tab) and *goodbye.java*.

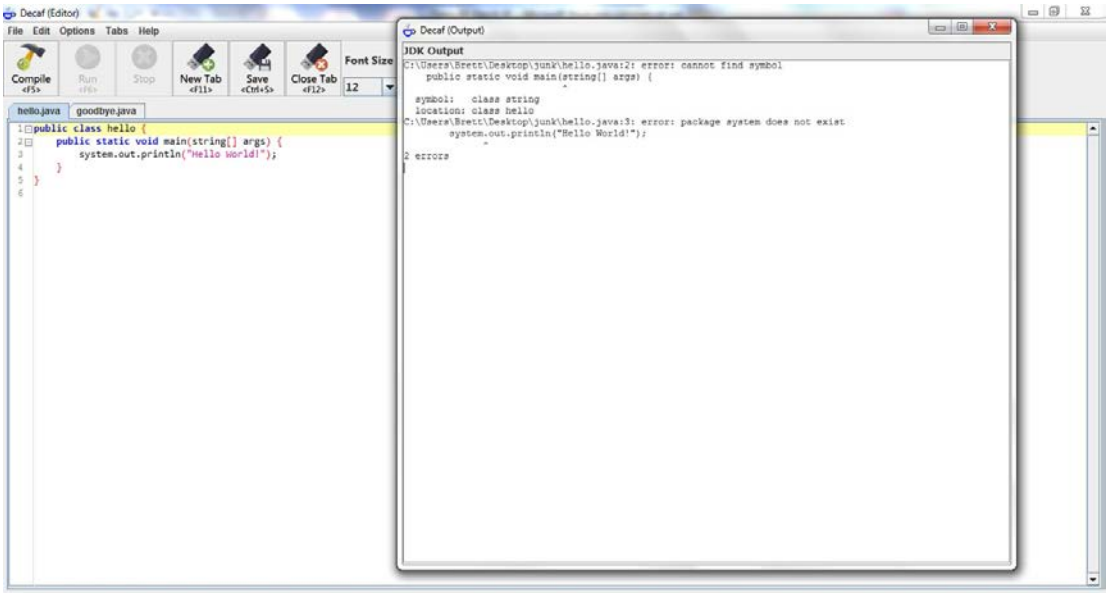

Figure C.3 Decaf seen in 'pass through' mode, where CEMs are not enhanced, but passed straight on to the user. Here two CEMs have occurred, *cannot find symbol* and *package system does not exist*.

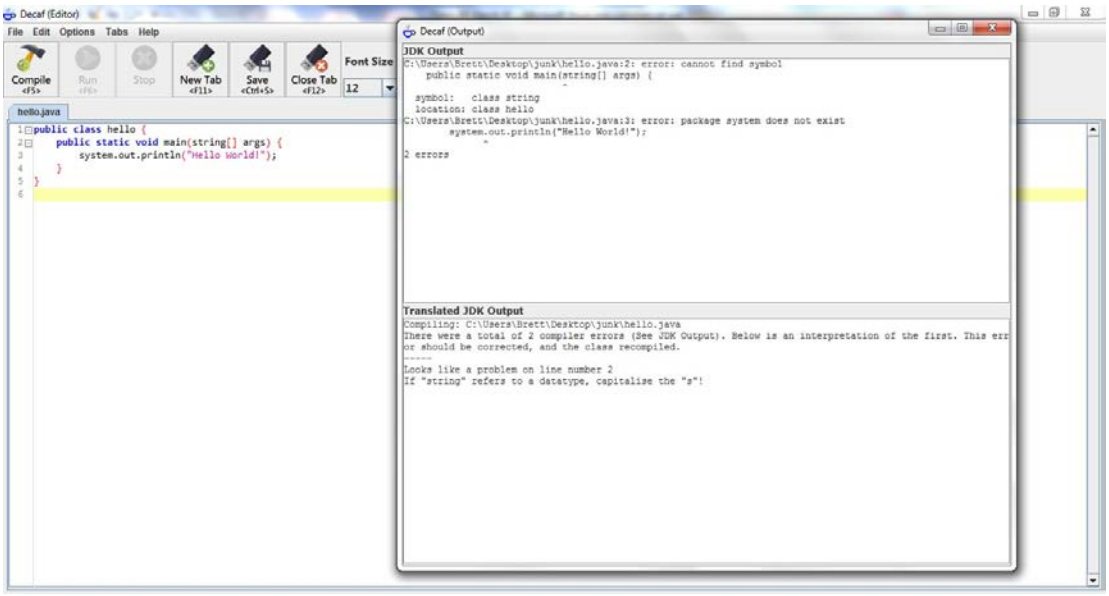

Figure C.4 Decaf in regular (enhanced) mode. Two raw CEMs are presented at the top, while the first is enhanced and presented below.

Appendix D Designing the Enhanced Compiler Error Messages

The principal consideration that influenced the design of Decaf was that the Java compiler error messages could, and should, be improved upon. This was inspired by the work of Michael Kölling, the developer of BlueJ, who said of error messages (1999, pp. 145-146):

> *Good error messages make a big difference in the usability of a system for beginners. Often the wording of a message alone can make all the difference between a student being unable to solve a problem without help from someone else and a student being able to quickly understand and remove a small error. The first student might be delayed for hours or days if help is not immediately available (and even in a class with a tutor it may take several minutes for the tutor to be able to provide the needed help).*

The design of Decaf took the clear presentation of error messages first and foremost. Pane and Myers (1996) highlighted 'simple and useful error messages' as one of the requirements of programming environments for novices, something which Nielsen also agreed with: 'Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution' (1994, p. 30). Lang (2002) also noted that compilers should not be exempt from user interface considerations, and that good error messages are important, especially for new users – compiler writers should consider their audience when designing error messages. Further, compiler writers should consider the audience for error messages and decide whether they are better aimed at the experienced programmer (who probably does not need them), or the novice (who certainly does). Simple vocabulary was recommended by (Marceau, Fisler, & Krishnamurthi, 2011a).

The above only offer general direction at best. Traver (2010) provided eight principles of good error message design. Traver's paper utilised examples of C++ SEMs to illustrate the above principles, and as Java is a C-like language, these were translated directly into practical advice for writing Decaf's enhanced SEMs. These principles, followed by heuristics (in brackets) are presented below, each followed by examples of how they guided the design of Decaf's ESEMs.

1. **Clarity and brevity (aesthetic and minimalist design, recognition rather than recall)**

The fact that Java SEMs are notoriously cryptic and terse, focus was put on clarity, while keeping brevity in mind. It is noted that Nianaltowski, Pedroni and Meyer's finding shows that more detailed messages do not necessarily simplify error interpretation (2008), however their study was for multiple languages including Java, but did not incorporate 'long form' Java messages – in fact of the five languages studied, only one had long form SEMs.

2. Specificity (recognition rather than recall; help user recognize, diagnose and recover from errors).

The largest problem relating to specificity is the lack of a one-to-one mapping for some Java errors. In cases where a particular SEM has multiple syntactical causes (an example of the lacking of one-to-one mapping discussed in Section 2.3), further program logic attempts to determine the specific cause of the error by analysing the offending line of user code. One such example is the error 'cannot find symbol'[20]. Ben-Ari (2007) notes that this error can be caused by inconsistencies between the declaration of an identifier and its use, a non-exhaustive list of syntax issues resulting in such is:

d. misspelled identifier (including capital letters used incorrectly)
e. calling a constructor with an incorrect parameter signature
f. using an identifier outside its scope.

An example of a line of code giving rise to this error (which every novice Java program must include correctly even in the most basic of programs) is the declaration of the main method:

```
public static void main(string[]args)
```

The javac error returned is:

```
test.java:4: error: cannot find symbol public static void
main(string[]args)
     ^
symbol: class string
location: class test
```

When Decaf intercepts this error, it is marked as one with multiple probable causes, and program logic determines that it is type a. above and the 's' in 'string' needs to be

[20] This is one of the most common (and notorious) Java SEMs, and will come up again.

capitalised. The complete error pair presented to the user contains the original javac error (above) and the following enhanced error:

```
Looks like a problem on line number 4.
If "string" refers to a datatype, capitalise the "s"!
```

Another issue relating to specificity is SEMs which are (notoriously) too general, such as:

```
not a statement
```

When Decaf encounters this SEM, it alerts the user to several possible issues from which it can arise:

```
1. Check indicated line for misspellings.
2. If a method is being called, make sure that the number
   and types of arguments are correct. If the method has
   no arguments, make sure that empty parenthesis '()'
   appear after method name.");
3. Check that no variable names start with numbers or
   other disallowed characters.
4. Check that you did not use == where you meant to use =
5. Check that you did not use + = instead of +=
6. Also check for a stray semicolon")
```

3. Context-insensitivity (consistency and standards)

This often occurs when the same logical error originates the same message, regardless of the context. An example of this is the SEM:

```
illegal start of expression
```

When Decaf encounters this SEM, it alerts the user to several possible issues, depending on the context from which the error originates (in this case possibly a switch statement, or a method within a method):

```
Check the following:
  1.  Did you type something like x + = 1 instead of x +=
1
  2.  If in a switch statement, make sure you did not type
'case: something' instead of 'case something:'
  3.  Make sure you are not writing a method inside
another method.
```

4. Locality (flexibility and efficiency of use).

This refers to a discrepancy between the true origin of the error and where the message indicates the error is. For instance, Java will report the following SEM, with (as always) a line number which is supposed to indicate where the actual error is:

```
class, interface or enum expected
```

However, this particular SEM can occur due to several issues, in many areas of the code. Decaf tries to address this with the following:

```
Make sure that a package statement (if needed) is before
any import statements.
Alternatively, check your class access modifier(s).
```

5. Proper phrasing (match between system and the real world).

This principle has four guidelines, which Decaf tries to follow:

i. Positive tone – avoid *illegal, invalid, bad, etc.*

   Decaf follows this, specifically under the guidelines of Shneiderman (1982, p. 610): “have a positive tone indicating what must be done, rather than condemning the user for the error. Reduce or eliminate the use of terms such as "ILLEGAL", "INVALID", "ERROR", or "INCORRECT".

ii. Constructive guidance – provide guidance in what to do

   For instance, when the SEM

   ```
   <identifier> expected
   ```

   is encountered, Decaf responds:

   ```
   Make sure that your method declaration(s) are not 'void
   static'. A static and void method must be declared
   'static void'.
   Have you entered something without declaring what type it
   is? For instance, did you write the name of a variable
   but not indicate a type on line 21?
   ```

iii. Programmer language – use language appropriate to the user’s level of knowledge – in this case *novice programmer*.

iv. Nonanthropomorphic messages – avoidance is advised, however this deserves further investigation (Traver, 2010).

6. Consistency (consistency and standards)

Decaf’s ESEMs attempt to be consistent, for example: if alternate actions are suggested, this should always be done in the same way; ESEMs are always of approximately the same length; etc.

7. Suitable visual design (aesthetic and minimalist design; error prevention).

Although Traver suggests that the importance of this principle should not be underestimated, he admits that it is a ‘higher-level’ issue (2010, p. 10). Decaf presents ESEMs in a simple, consistent manner similar to other successful pedagogic programming environments.

8. Extensible help (help and documentation)

Traver explains: help provided by messages could be organized into levels: a first short message would probably be enough most of the time; if not, some brief explanation or examples can give extra assistance to the programmer. A further level could consist of a list of potential corrective actions. As stated above, most of Decaf's ESEMs are brief and are probably enough. Some provide more guidance; sometimes through example (see 5ii above). Still others provide more 'extra assistance', such as when the SEM

```
cannot variable cannot be referenced from a static context
```

is encountered, Decaf provides the following ESEM (which is an exception, providing a 'further level':

```
You have happened upon one of the most common, yet most
difficult to understand errors in beginner-level Java
programming. It is well worth reading this brief
explanation:
http://www.coderanch.com/t/606337/java/java/main-method-
static-access-static
```

This is currently the only Decaf ESEM which uses an external reference for guidance. Although referring students to 'extra' or supplemental help has been done before, it was found that students may not utilise extra references (Hristova, Misra, Rutter, & Mercuri, 2003), read messages fully (Schorsch, 1995) or utilise extra help (Thompson, 2004). This was offered as a potential explanation for the negative result of (Denny, Luxton-Reilly, & Carpenter, Enhancing syntax error messages appears ineffectual, 2014). In the case of this particularly common and in my experience notoriously difficult error, an external reference was deemed warranted.

Appendix E Categorising the Decaf Software

Despite a relative lack of information on how to *design* systems for novices, Kelleher and Pausch (2005) provided a taxonomy for them. The closest category for Decaf in this taxonomy is *3. Teaching Systems > 3.1 Mechanics of Programming > 3.1.1 Expressing Programs > Simplify Entering Code > 2. Prevent Syntax Errors*. Although Decaf does not attempt to explicitly prevent syntax errors – for instance there is no mechanism to prevent or deter students from entering syntactically invalid code – it is a goal of Decaf to prevent syntax errors by providing students with useful information to resolve them. This is intended to help students learn to prevent them in the future. Kelleher and Pausch conducted further investigation by looking at the attributes of the systems they classified. They recognised that each system appeared in their taxonomy only once, and that some may have been built on the lessons of preceding systems. They sought to uncover the major design influences of the systems they categorised, including those that were not the primary contribution of a given system. In doing so they addressed nine questions, the sixth of which was "Does the programming environment attempt to prevent syntax errors in any way?" Under the umbrella attribute of *Preventing Syntax Errors*, five attributes were investigated, including providing *better syntax error messages*. Every system they categorised (regardless of the category a system was placed in) was checked to see if 'better syntax errors' was an attribute of the system. Only JJ (Motil & Epstein, n.d.) was found to do so, appearing in the category *3. Teaching Systems > 3.1 Mechanics of Programming > 3.1.1 Expressing Programs > Simplify Entering Code > 1. Simplify Entering Code*, a sister category to *2. Prevent Syntax Errors.*

Appendix F All Compiler Error Messages Logged

CEM number	Enhanced by Decaf?	CEM description
1	yes	'(' expected
2	yes	'(' or '[' expected
3	yes	')' expected
4	yes	'.' expected
5	-	'.class' expected
6	-	: expected
7	yes	';' expected
8	yes	'[' expected
9	yes	']' expected
10	yes	'{' expected
11	yes	'}' expected
12	yes	<identifier> expected
13	-	> expected
14	-	-> expected
15	-	array dimension missing
16	yes	array required, but *type* found
19	yes	bad operand type *type_name* for unary operator '*operator*'
20	yes	bad operand types for binary operator '*operator*'
22	-	break outside switch or loop
23	-	cannot assign a variable to final variable *variable_name*
24	yes	cannot find symbol
25	-	cannot return a value from method whose result type is void
27	-	'catch' without 'try'
29	yes	class *class_name* is public, should be declared in a file named *class_name*.java
31	-	class expected
32	yes	class, interface, or enum expected
34	-	constructor *constructor_name* in class *class_name* cannot be applied to given types;
36	-	double cannot be dereferenced
38	-	duplicate class: *class_name*
39	-	'else' without 'if'
40	-	empty character literal
43	-	exception *exception_name* is never thrown in body of corresponding try statement
46	-	illegal '.'
47	yes	illegal character: '*character*'
48	-	illegal escape character
49	-	illegal initializer for *type*
50	-	illegal line end in character literal
51	yes	illegal start of expression
52	-	illegal start of statement
53	-	illegal start of type
54	-	illegal static declaration in inner class *class_name*
55	-	illegal underscore

56	-	incomparable types: *type* and *type*
57	yes	incompatible types OR incompatible types: *type* cannot be converted to *type*
58	-	inconvertible types
59	-	*type* cannot be dereferenced
60	-	integer number too large: *value*
61	yes	invalid method declaration; return type required
63	-	malformed floating point literal
64	-	method *method_name* in class *class_name* cannot be applied to given types;
65	-	method *method_name* is already defined in class *class_name*
66	-	missing method body, or declare abstract
67	yes	missing return statement
69	-	modifier static not allowed here
70	-	no suitable constructor found for *method_name*
71	-	no suitable method found for *method_name*
72	-	non-static method *method_name* cannot be referenced from a static context
73	yes	non-static variable *variable_name* cannot be referenced from a static context
74	yes	not a statement
77	yes	package *package_name* does not exist
78	yes	possible loss of precision
79	-	reached end of file while parsing
81	-	repeated modifier
83	yes	'try' without 'catch', 'finally' or resource declarations
85	-	unclosed character literal
86	yes	unclosed comment
87	-	unclosed string literal
89	yes	unexpected type
90	-	unreachable statement
91	yes	unreported exception *exception_type*; must be caught or declared to be thrown
92	yes	variable *variable_name* is already defined in method *method_name*
93		variable *variable_name* might not have been initialized
94	-	'void' type not allowed here
95	-	while expected

Appendix G Interview Questions and Comments

Below I present the nine interview questions, followed by selected comments made by the lecturer. Comments from the control group interview are numbered C*x* and from the intervention interview I*x*.

Question 1: You used Decaf with students for six weeks. What was that experience like?

C1: The students used Decaf very similarly to other editors that I have used before. There were a few small issues getting started but that is normal.

C2: Some students took more time than others to adjust to the software but no more than other IDEs [editors].

I1: The students seemed to enjoy using Decaf. I noted on a few occasions that they were comparing their error messages and asking each other questions like 'what did Decaf tell you for error xxx'.

I2: In labs I spent more time with students in groups than normal. It was informal – two, three, maybe four students all looking at one student's screen.

I3: I got a lot of questions of the causes of syntax errors, especially where more than one type of programming mistake can cause the same [syntax] error.

Question 2: How did you observe the student experience with Decaf?

C3: I noticed that there were students that chose to use other IDEs, maybe three or four. That is pretty normal though. I think they were using Eclipse. I was wondering if more students would move towards Eclipse but I did not notice that.

I4: Students started to name Decaf when asking me questions. Such as 'Decaf gave me this error, what do I do.' Normally students say 'Java gave me this error' no matter what the IDE they are using is. It seemed like the students really embraced Decaf. They saw Decaf as separate to the language.

Question 3: What did you notice in terms of the student experience with decaf?

C4: There was a good bit of frustration with syntax. They were very focussed on it for the first few weeks. This is common though.

I5: Students were very positive about Decaf. They knew that it was written by you [Brett] at CCT and they asked about it. They were impressed or intrigued that the software was written for them.

I6: Many students continued to use Decaf after six weeks. They are still using it.

Question 4: Did you notice anything about student compiling behaviour with Decaf?

C5: Like I said, there was a lot of concern about syntax.

I7: I think I noticed more group work. There was definitely more interaction in the lab than I expected. This could have been the group, but the lab work was very collaborative this year.

Question 5: How did this differ from your previous experiences teaching programming?

C6: It was a pretty normal semester.

I8: The transition from Decaf to EJE was time consuming. Normally students get used to a new IDE pretty quickly, within a week. But I was getting questions on basic use of EJE for probably two weeks.

I9: It was more interactive. I had a lot of questions this year. Normally between lecture and lab there are few questions. This year I might spend 15 or 20 minutes between lecture and lab answering questions.

Question 6: Once EJE was introduced, did you notice any changes?

C7: The switch to EJE was smooth.

C8: I noticed that a good number of students kept using EJE.

I9: A lot of students kept using Decaf.

I10: I noticed several students using EJE and Decaf which was interesting. One student used both, copying and pasting code back and forth. When I asked him about this he said that if he has trouble with EJE he would see if Decaf would explain an error better.

Question 7: Did you notice any drawbacks of using Decaf?

C9: Decaf was giving some students problems with non-English keyboard layouts. I have noticed the same problems with EJE also though.

I11: One week several students had trouble with Decaf and infinite loops. I think I know what the problem is… The students were intentionally programming infinite loops to see if they could figure out what the problem was. They were very interested.

Question 8: Did you notice any positive effects of using Decaf?

C10: It worked well. Having a minimal amount of options was a huge benefit. Normally I get several questions per semester about different options in Eclipse. With Decaf the students use all of the features, because there are so few. But they do not need more options this early on.

I12: The students were very engaged this semester. Many lectures were very discussion based. I had a lot of very good questions.

I13: I would say I got less questions on 'how do I make this error go away' and more questions on 'why is this error happening'. They were more focused on solutions than just fixing and forgetting.

Question 9: What would you recommend to improve either Decaf itself, or using it in the classroom with students?

C11: I know I'm supposed to only talk about the control group, but I think things would have been smoother, and maybe more interactive this year if they had the full Decaf version [The version which enhances CEMs, that the intervention group had].

C12: We need to fix the non-English keyboard layout issue.

C13: There was also an install problem with Mac computers. I was able to get around it but it is something we have to do.

I14: The infinite loop issue needs to be fixed, but it does not come up often at all. I think one student happened on it and told a few others. It was not a big problem.

Appendix H Runtime Errors

This work is primarily concerned with compile-time errors (syntactic and those semantic errors which are caught by javac), however it should be noted that many IDEs also report runtime errors. Although runtime error messages may look like those generated by bad syntax (and sometimes bad semantics), they happen during program execution and never occur due to syntax errors. Murphy et al. (2008) developed a tool which enhanced runtime errors in Java, for novice use.

Decaf provides enhanced error messages for the following runtime errors, in a manner very similar to the ECEMs it provides.

- java.lang.ArrayIndexOutOfBoundsException
- java.lang.NullPointerException
- java.lang.ArithmeticException: / by zero
- java.lang.StringIndexOutOfBoundsException
- java.util.InputMismatchException
- FileNotFoundException
- NumberFormatException